CUSTOMER
NUMBER 22852

FINNEGAN, HENDERSON, FARABOW, GARRETT & DUNNER, L.L.P.
1300 I STREET, N. W.
WASHINGTON, DC 20005-3315

202 • 408 • 4000
FACSIMILE 202 • 408 • 4400

ATLANTA
404 • 653 • 6400
PALO ALTO
650 • 849 • 6600

WRITER'S DIRECT DIAL NUMBER:
(202) 408-4039

TOKYO
011 • 813 • 3431 • 6943
BRUSSELS
011 • 322 • 646 • 0353

August 11, 2000

ATTORNEY DOCKET NO. **07030.0011-00**

Box PATENT APPLICATION
Assistant Commissioner for Patents
Washington, D.C. 20231

Re: New U.S. Patent Application
Title: HIGH-PERFORMANCE RISC-DSP
Inventor(s): William DALLY et al.

Sir:

We enclose the following papers for filing in the United States Patent and Trademark Office in connection with the above patent application.

1. Application - 38 pages, including 5 independent claims, 10 claims total and an Abstract.

2. Drawings - 33 sheets of informal drawings (Figures 1-17(h)).

3. A check for $846.00 representing a $690.00 filing fee and $156.00 for additional claims.

Applicants claim the right to priority based on Provisional Patent Application No. 60,148,652 filed August 13, 1999.

This application is being filed under the provisions of 37 C.F.R. § 1.53(f). Applicants await notification from the Patent and Trademark Office of the time set for filing the Declaration.

Please accord this application a serial number and filing date.

The Commissioner is hereby authorized to charge any additional filing fees due and any other fees due under 37 C.F.R. § 1.16 or § 1.17 during the pendency of this application to our Deposit Account No. 06-0916.

Respectfully submitted,

By: _____
E. Robert Yoches
Reg. No. 30,120

ERY:bbp
Enclosures

Attorney Docket No. 07030.0011

United States Application

of

William J. Dally, Staffan Ericson, W. Patrick Hays, Robert Gelinas,
Sol Katzman, and Sam Rosen

for

HIGH-PERFORMANCE RISC-DSP

I. RELATED APPLICATIONS

This application relates to provisional application Serial No. 60/148,652 filed on August 13, 1999, drawing priority therefrom.

II. BACKGROUND OF THE INVENTION

The present invention relates to the field of digital signal processor (DSP) architectures, and more particularly to DSP architectures with optimized architectures.

With the increasing commercial importance of DSP-intensive applications, such as wireless communication, modems, and computer telephony, has come an increasing recognition of the benefit of implementing DSP functions on a CPU. Not only are CPUs usually needed for memory management, user interface and Internet Protocol software, CPUs also have excellent third-party software tool support.

Implementing certain DSP algorithms, such as the FIR Filter or Discrete Cosine Transform (DC7), in software, however, may degrade performance up to an order of magnitude as compared to specialized DSPs. Another difficulty is the problem of deterministic real-time allocation in sophisticated CPUs.

Some vendors have tried to address these problems by offering auxiliary processing components. DSP coprocessors, for example, use separate instruction sets, instruction stores and execution units, and DSP accelerators share the same I-stream with the CPU but have separate execution units. These approaches, however, impose a substantial burden on the CPU in managing DSP functions.

III. SUMMARY OF THE INVENTION

Systems and methods consistent with the present invention provide for an alternate DSP architecture configuration that allows for more efficient implementation of Dsp algorithms and use of CPU resources.

A digital signal processor consistent with this invention comprises two execution pipelines capable of executing RISC instructions; instruction fetch logic that simultaneously fetches two instructions and routes them to respective pipelines; and control logic to allow the pipelines to operate independently.

Another digital signal processor consistent with this invention and capable of integrating subopcodes into an established CPU instruction set comprises a memory that stores instructions having opcodes; an instruction decoder that identifies a relocatable opcode to designate subopcodes; and a subopcode detector that decodes subopcodes if the instruction decoder identifies the relocatable opcode.

Still another digital signal processor consistent with this invention comprises a register pair; and means for executing a multiply instruction on a number stored in the register pair, including first means for performing multiply instructions on higher-order portions of each register in the register pair, second means for performing multiply instructions on the remaining portions of each register in the register pair, and third means for combining the results from the first and second means.

A circular buffer control circuit consistent with this invention comprises a first number of circular buffer start registers; a first number of circular buffer end registers, each associated with a different one of the circular buffer start registers; and circular buffer control logic including

means for comparing a pointer to and address in a selected one of the circular buffer end registers, and means for restoring the address in the one of the circular buffer start registers associated with the selected circular buffer end register if the pointer matches the address in the selected circular buffer end register.

IV. BRIEF DESCRIPTION OF DRAWINGS

The accompanying drawings, which are incorporated in and constitute a part of this specification, illustrate one embodiment of the invention and, together with the description, serve to explain the objects, advantages, and principles of the invention. In the drawings:

Fig. 1 is a block diagram of a DSP architecture consistent with this invention;

Fig. 2 is a data flow diagram of the superscalar instruction issue consistent with this invention;

Fig. 3 is a table indicating the instruction select logic consistent with this invention;

Fig. 4 shows a superscalar RALU datapath consistent with this invention;

Fig. 5 shows an example of a MMD register consistent with this invention;

Fig. 6 is an illustration of a dual MAC datapath consistent with this invention;

Figs. 7A, 7B, and 7C show some of the data arithmetic modes;

Figs. 8A, 8B, 8C, and 8D contain a table of the instructions supported by an implementation consistent with this invention;

Fig. 9 is a table showing the assignment of instructions to different pipelines;

Fig. 10 is a block diagram of circular buffers consistent with this invention;

Figs. 11A, 11B and 11C show a table summarizing vector addressing instructions;

Figs. 12A and 12B show a table with instructions when a saturation option is provided;

Figs. 13A and 13B show a table with additional ALU operations;

Fig. 14 contains a table with conditional operations;

Fig. 15 contains a table with the cycles required between instructions;

Fig. 16 is a block diagram of several coprocessor registers; and

Figs. 17A-H illustrate additional LEXOP codes.

V. DESCRIPTION OF THE PREFERRED EMBODIMENTS

Reference will now be made in detail to embodiments consistent with this invention that are illustrated in the accompanying drawings. The same reference numbers in different drawings generally refer to the same or like parts.

A. OVERVIEW

Performance critical DSP algorithms such as FIR filters need to perform multiply-accumulate on memory-based operands. Typically, DSP architecture have a CISC instruction set with memory-based operands. RISCs on the other hand have load/store architectures and register operands. Extending the RISC into DSP with superscalar issue allows one instruction to load the register from memory and another to execute on register-based operands that were loaded earlier.

A RISC-DSP consistent with the present invention can achieve a system clock speed of 200 MHZ and a peak computational power of 400 million multiply-accumulate (MAC) operations per second when implemented in 0.18 μm technology, which is comparable to the highest-performance DSPs available. Also, tightly integrating DSP extensions into an existing instruction set, such as the MIPS ISA, gives access to a wide variety of third-party tools and allows programmers to switch seamlessly from RISC code to DSP code. The extensions include

(1) multiply-accumulate (MAC) instructions, including guard bits (with support for fractional arithmetic mode), and saturation and rounding for high-fidelity DSP arithmetic operations that have not previously been available in RISC processors, (2) dual 16-bit versions of all ALU operations, (3) post-modified memory pointers with hardware circular buffer support, (4) zero-overhead loop counter, which can be interrupted and nested, (5) conditional move, (6) specialized ALU operations, (7) prioritized low-overhead interrupts, and (8) load/store of two 32-bit general registers with a single instruction.

These ISA extensions are accomplished by introducing an I-Format opcode called LEXOP, which creates 64 additional subop codes encoded in INST[5:0].

Fig. 1 shows an example of a DSP 100 architecture consistent with this invention. DSP 100 includes a Coprocessor 0 (CP0)110, a Register File Arithmetic Logic Unit (RALU) 120, instruction memory and issue logic 130, data memory 140, and LBC 150. IADDR (instruction address b) bus 160 links coprocessor 0 110, instruction memory and issue logic 130, and LBC 150. DADDR (data address) bus 170 links RALU 120, data memory 140, and LBC 150. Instruction pathways INSTA 163 and INSTB 166 connect between CP0 110, RALU 120, instruction memory and issue logic 130, and LBC 150. DBUS (data bus) 175 connects between CP0 110, RALU 120, data memory 140, and LBC 150.

CI (customer interface) 180 acts as an interface to a customer coprocessor and connects to INSTA pathway 163 and DBUS 175. LBC 150 also interfaces to DI (data in) bus 192, DO (data out) bus 194, address bus 196, and control bus 198.

One implementation of the DSP consistent with this invention adds a stage to a dual-issue, five-stage pipeline to form dual-issue six-stage pipelines. The additional stage is a D

(Decode) stage. This superpipeline design can achieve higher performance and isolate the processor's logic from customer-supplied instruction memories. Sustained performance near the peak 400 million MACs per second can be realized for inner loops of key DSP algorithms.

One pipeline is preferably a load/store pipe. It includes data memory access and all instructions except multiply and divide operations. Another pipeline is preferably a multiply-accumulate pipe. It includes a MAC and ALU, each with dual 16-bit operations.

Preferably, DSP algorithms will use the load/store pipe to load a pair of operands into a general register while executing dual MAC operations in multiply-accumulate pipe on earlier data. Decoupling register loads from the MAC processing allows loop unrolling and takes effective advantage of the thirty-two general registers for temporary storage. In addition, dual-issue allows the DSP to achieve the memory bandwidth required by DSP within a RISC architecture, and dual 16-bit SIMD operations take full advantage of the 32-bit RISC datapaths.

B. Description of Key Components and Features

1. Superscalar Architecture

As explained above, a DSP consistent with this invention preferably issues dual 32-bit instructions to two distinct six-stage execution pipelines. Fig. 2 shows data paths for the instruction issue. An instruction cache and IRAM 210 in instruction memory and issue logic 130 (Fig. 1) can fetch two 32-bit instructions simultaneously. Following the superscalar instruction buffer and issue logic described below, the instructions issue as IB_S_R to one pipeline, called Pipe B and IA_S_R to the other pipeline, called Pipe A. Pipe A would be the "load/store pipe" described above because it uniquely supports the load and store operations. Pipe B, then, would be the multiply-accumulate pipe.

The dual-issue design offers significant performance advantages. For example, peak computational performance for DSP algorithms typically requires at least one memory operand per instruction cycle. The dual-issue superscalar design, on the other hand, allows a memory operand to be loaded by one instruction while the MAC operates on earlier register-based data.

The RISC data path is 32-bits, but few DSP algorithms require more than 16-bits of precision. This allows the simultaneous fetching of two values from memory, which further improves performance.

The six stages of the execution pipelines are as follows:

Stage 1	I	Instruction fetch
Stage 2	D	Decode
Stage 3	S	Source fetch
Stage 4	E	Execution
Stage 5	M	Memory data select
Stage 6	W	Write back to register file

To avoid degrading performance, the superscalar issue logic can operate during the Decode-Stage of the pipeline that occurs after I (Instruction Fetch) and before S (Source Fetch). The D stage allows a better system clock cycle time over that of a five-stage pipeline. Support for fully synchronous instruction memories also isolates the processor logic from the customer-supplied memories, which facilitates integration.

Although the D-Stage incurs a two-cycle penalty on branch prediction failure as compared to the one-cycle penalty from a five-stage pipeline, the effect of this penalty does not usually manifest. For example, because branches are predicted taken in all cases, no wasted

cycles are incurred for backward branches used as loop counters except in the final cycle where the loop "falls through."

A two-bit Valid register 220 (V0 and V1) is updated with each fetch to indicate whether instructions I0 230 and I1 235 are valid. The Valid register can indicate three states: (1) "neither valid," which occurs following a cache miss, (2) "both valid," which occurs following a cache hit; or (3) "V0=invalid/Vl=valid," which occurs following a branch to I1. Later, one or both instructions may issue to Pipes A and B. If only one instruction from a valid pair (*e.g.*, I0) issues, Valid register 220 will be appropriately updated and instruction fetch will be stalled while the second instruction issues. V0=invalid/V1=valid occurs following a branch to Il. V0=valid/ Vl=invalid will not occur because preference is given to I0 (the first instruction in program order) if only one instruction can issue. The next instruction Il issues before fetching a new pair of valid instructions, I0/II.

Decoding register 220 belongs to instruction analysis logic 240, which, along with the Instruction Select logic, is implemented in the D-Stage. Instruction analysis logic 240 generates five key signals: 0eA, leA, 0eB, leB. 0eA indicates whether I0 can execute in Pipe A. This decision is based on the I0 opcode and the resources available in Pipe A. For example, if I0 is an ADD and Pipe A can execute an ADD, then 0eA = 1; if I0 is a MAC and Pipe A has no MAC then 0eA = 0. The complexity of this logic can minimized by careful encoding operations and by careful partitioning of operations between the two Pipes. One simplification could be to partition the DSP so that the Pipe B instruction, IB_S_R, is routed only to RALU 120 (Fig. 1).

A special signal ld0 indicates that both I0, I1 are valid and Il (the higher logical order) and depends on I0. This will occur if the result of I0 is used as a source by I1. In this case, only I0 will issue. Analysis of the other signals is shown below:

Signal	Meaning
0eA	I0 is valid, I0 can be executed in Pipe A
1eA	I1 is valid, I1 can be executed in Pipe A
0eB	I0 is valid, I0 can be executed in Pipe B
1eB	I1 is valid, I1 can be executed in Pipe B
1d0	I0, Il are valid and Il depends on I0: - source of Il = dest of I0, or - I1 updates register file and I0 post-modifies load/store pointer

Based on the Instruction Analysis, instructions are issued to Pipes A and B. Three outcomes are possible for each Pipe,

IA(IB) ← I0

IA(IB) ← I1

IA(IB) ← NOP

Data flow is effected by input multiplexers 260, 262, instruction registers 264, 266, output multiplexers 268, 270, and output registers 272, 274. An input multiplexer 276 and an output register 278 are associated with Valid register 220.

Instruction select logic 250 controls output multiplexers 272, 274 according to Table 300 in Fig. 3. Table 300 also specifies the update to the Valid register as a consequence of the issue decision.

"V ← next" in Table 300 indicates that I0/I1 are updated with a new instruction pair (following cache load) and both are valid, unless a branch to an odd address occurs. In that case, I0 is not valid and I1 is valid. If two instructions are valid, but only one instruction issues, the signal "Stall" will be active to stall the instruction fetch by one cycle until the 2nd instruction has issued.

The Order Register ORD, which lies in instruction memory and issue logic 130 (Fig. 1) indicates which instruction is first in program order. For example if IA ← I1 and IB ← I0, then ORD ← "B" to indicate that the instruction in Pipe B is first in program order. ORD must be examined by later-stage control in two cases. First, if both instructions attempt to write the same general register, ORD will select the latter in program order. Second, if both instructions generate exceptions, ORD will select the address of the earlier instruction into the Exception PC (EPC) register. This logic is preferably executed during the D-Stage. IB_S_R, IA_S_R and ORD_S_R issue from registers on the rising edge of the S-Stage clock cycle, and are broadcast to other sections.

To improve performance, the DSP's superscalar implementation can use a "sliding" two-instruction buffer with a third instruction register, I2. If only one instruction (I0) issues, the contents of Il are shifted into I0, the contents of I2 are shifted into Il and a new instruction is fetched into I2. As a result, a pair of instructions will always be available in I0 and Il for potential dual issue, except for the cycle following a branch or jump.

The third instruction buffer register I2 is required because another pair of unaligned instructions must be available for use, following the dual issue of an unaligned instruction pair (I0 and I1 addresses are not both on the same factor-of-8 boundary). If memory fetches only aligned instruction pairs, a third instruction buffer is required.

The compression signal indicates whether code compression has been enabled. When compression is enabled, each 32-bit fetch is interpreted as a pair of 16-bit instructions. Upon fetching I0(32), the first 16-bit instruction of the pair will be loaded into I0 REG and the second into Il REG. Similarly, when Il(32) is fetched, the first instruction of the pair will be loaded into I0 REG and the second into Il REG. D-Stage logic follows the approach previously described with the shorter instruction (*e.g.*, 16-bit) opcodes being analyzed for potential issue, then selected into IB REG and IA REG. The critical Register File read addresses for shorter instruction opcodes are resolved during the D-stage so that register file access for shorter instructions, as for longer (*e.g.*, 32-bit) instructions, can begin on the rising edge of the S-Stage clock. The state of I0 valid and I1 invalid does not occur because there is no out-of-order execution.

2. RALU Datapath

Fig. 4 illustrates the Superscalar RALU datapath 400 consistent with this invention. Operations are divided between Pipe A and Pipe B in such a way that RALU 120 (Fig. 1) is the only major section of the processor which requires both Pipe A and B instructions. Coprocessor 0 110, as well as the customer-defined coprocessors 1-3 (not shown), only require the Pipe A instruction.

An 8-port (4-read/4-write) general register file 410 supports the dual execution pipelines. The 8-port architecture allows a large portion of both pipes to be fully replicated, simplifying design and improving likelihood of two instructions being able to dual issue.

As Fig. 4 shows, ALU A 420 and ALU B 430 connect to register file 410 through the respective read and write ports. In each pipe, one write port is dedicated to register file updates from the data bus (*e.g.*, loads, or MFCz, CFCz - moves from a coprocessor). The remaining three ports (two read ports and one write port) are available for the other operations assigned to that Pipe. As a result, loads, including "twinword" loads of register pairs can dual-issue with any MAC or ALU instruction assuming no data-dependency. The nomenclature "twinword" distinguishes these operations from "doubleword" operations which (in other extensions) access a single 64-bit general register. The "twinword" load/store (Load/Store of pairs of 32-bit registers (e.g., r16, r17) in one 64-bit transfer from memory) allow the dual MACs to be kept busy at all times.

All ALU operations are available in both Pipe A and Pipe B. DSP extensions to memory addressing, such as pointer post-modification and circular buffer addressing described below, are preferably unique to Pipe A. Also, coprocessor operations and all "sequencing control instructions" (branches, jumps) are unique to Pipe A. As a result, Pipe B instructions are not routed to the coprocessors. This is shown in Fig. 4 with ALU B being connected to Custom Engine Interface (CEI) 450 and dual MAC 44, which preferably resides in RALU 120 (Fig. 1). CEI 450 is optionally available for customer proprietary operations only in Pipe B. This feature allows the customer extensions to maintain high throughput since they can dual-issue with Load and Store instructions which issue to Pipe A.

3. MMD register

Fig. 5 shows MAC Mode ("MMD") register 500, which is preferably a new Radiax User register (24) that is accessed using Radiax User Move instructions MTRU and MFRU. MMD register 500 is read using the MFLXC0 instruction, a variant of the MFC0 instruction and is loaded using a MTLXC0 instruction, a variant of the MTC0 instruction. Field 510 [31-5] are the mask bits for eight low-overhead interrupts described below. Field MF 520 selects arithmetic mode for multiplies in the Dual Mac. A "0" means use integer arithmetic mode, and a "1" means use fractional arithmetic mode.

Field MS 530 selects saturation boundary in the Dual Mac accumulators as follows. A "0" means saturate at 40 bits, and a "1" means saturate at 32 bits. Field MT 540 selects truncation of 32x32 bit multiplies in the Dual Mac. A "0" means perform full 32x32 bit multiply (sum all four partial products), and a "1" means omit partial product rX[15:00] x rY[15:00] when performing 32x32 bit multiply.

Field RND 550 selects the rounding mode used in the *RNDA2* instruction. A "00" means convergent rounding, and a "01" means round to nearest number. In convergent rounding, which is sometimes called "round-to-nearest-even," the numbers are rounded to the nearest number. When the number to be rounded is midway between two numbers representable in the smaller format, round to the number is rounded to the even number. The rounded result will always have 0 in the lsb. If the lsb left of the roundoff point is random, convergent rounding is unbiased.

In rounding to the nearest number, the number is rounded to the more positive number when the number to be rounded is midway between two numbers representable in the smaller format. This rounding mode is common because it is easily implemented by always adding

0..0^10...0 to the number to be rounded. Digits to the right of "A" are dropped after rounding. Upon reset all bits in the MMD register 500 are initialized to 0.

4. MAC datapath

Fig. 6 illustrates a Dual MAC datapath 600 consistent with this invention. The major subsystems are two 16-bit multiply-accumulate datapaths 610 and 620, each with a temporary register 612 and 622, respectively, and divide unit 630. Each multiplier 610 and 620 feeds a corresponding 32-bit product register 614 and 624, two accumulator units 616 and 626, four 40 bit accumulator registers 618 and 628, and output scalers 619 and 629. A 40-bit Add/Subtract/Dual Round Unit 650 provides optional saturation and overflow for the two accumulators 616 and 626.

Multiply-accumulate datapaths can operate on 16-bit input data, either individually or in parallel. Preferably, the same assembler mnemonic is used for individual or parallel operation. The output register specified determines whether MAC0 or MAC1 or both, operate. For example,

MADDA2 m0*l*, r2, r3 // MAC0:	m0*l* ←m0*l* + r2[15:00] * r3[15:00]
// MAC1:	IDLE
MADDA2 m0h, r2, r3 // MAC0:	IDLE
// MAC1:	m0h ←m0h + r2[31:16] * r3[31:16]
MADDA2 m0, r2, r3 // MAC0:	m0*l* ←m0*l* + r2[15:00] * r3[15:00]
// MAC 1:	m0*l* ←m0*l* + r2[31:16] * r3[31:16]

Each multiplier 610 and 620 can initiate a new 16 x 16-bit product every cycle *(single cycle throughput)*. Each 16 x16-bit multiply-accumulate preferably completes in three cycles.

Temporary registers 612 and 622 and product registers 614 and 624 show that multipliers 610 and 620 require two cycles, but have single cycle throughput. Temporary registers 612 and

622, and product registers 614 and 624, however, are preferably not accessible by the programmer. Thus, there are two delay slots for multiplication or multiply-accumulate. For example,

```
Inst 1: MADDA2    m1h, r2, r3
Inst 2: delay slot 1              // new m1h is not available
Inst 3: delay slot 2              // new m1h is not available
Inst 4: MFA       r3, m1h         // new m1h is available
```

M1h can be referenced by MFA in Inst2 (Inst3), but two (one) stall cycles will be incurred. The number of stall cycles in DSP algorithms are expected to be minimal because many products are often accumulated before the accumulator output must be stored. In a 64-tap FIR, for example, sixty-four terms are accumulated before the filter sample is updated in memory. Also, the four accumulator pairs allow loops to be "unrolled" so that up to three additional independent MAC operations can be initiated before the result of the first is available.

Compared to a typical RISC multiply-accumulate unit, a MAC consistent with this invention includes a number of features critical to high-fidelity DSP arithmetic, such as accumulator guard bits, fractional arithmetic, saturation, rounding, and output scaling. These features are optionally selected by opcode and/or mode bits in MMD register 500, and are compatible with conventional integer arithmetic.

Figs. 7A, 7B, and 7C show some of the data arithmetic modes consistent with the present invention. These modes are used to select between several available options for the following features of the MAC's arithmetic: (i) truncation mode, (ii) saturation mode, (iii) fractional/integer

arithmetic mode, and (iv) rounding mode. These modes and their optional settings are discussed below.

Accumulation is preferably performed at 40-bit precision, using eight guard bits for overflow protection. The alternative is to require the programmer to right-shift (scale) products before accumulation, which complicates programming and causes loss of precision. Before accumulation, the product is sign-extended to 40-bits. With guard bits, the only loss of precision will typically occur at the end of a lengthy calculation when the 40-bit result must be stored to the general register file or to memory in 32-bit or 16-bit format.

Fractional arithmetic is implemented by the program's interpretation of the 16-, 32- or 40-bit quantities and is controlled by a bit in the MMD register 500. When fractional mode is selected, the dual MAC shifts the results of any multiply operation left by one bit to maintain the alignment of the implied radix point. Furthermore, since (-1) can be represented in fractional format but +1 cannot, in fractional mode the dual MAC detects when both operands of a multiply are equal to (-1). When this occurs, it generates the approximate product consisting of 0 for the sign bit (representing a positive result) and all 1's ones for the remaining bits. This is true for both l6xl6-bit and 32x32-bit multiplications. The least significant bit of a product is always zero in fractional mode (due to the left shift).

The accumulation units can add the product to, or subtract it from, one of the four accumulator registers 618 and 628. This operation can be performed with optional saturation; that is, if a result overflows (underflows), the accumulator is updated with the largest(smallest) positive (negative) number rather than the "wraparound" result with incorrect sign. The DSP instructions preferably include a multiply-add and multiply-sub instruction, each with and

without saturation. There are also instructions for adding or subtracting any pair of 40-bit accumulator registers together, with and without saturation. A bit in the MMD register determines whether the saturation is performed on the full 40-bits or whether saturation is performed at 32 bits. The latter capability is useful for emulating the results of other architectures that do not have guard bits.

When the instruction requires multiplication, but no accumulation, the product is passed through the accumulation unit unchanged. (Thus, both 16-bit multiplication and multiply-accumulate require three MAC cycles.)

A Round instruction can also be executed on one (or a pair) of the accumulator registers to reduce precision prior to storage. The rounding mode is selectable in MMD register 500. The output scalers 619 and 629 are used to right shift (scale) the accumulator register when it is transferred to the general register file.

The dual MAC configuration consistent with the present invention is also used to execute the 32-bit MULT(U) and DIV(U) instructions specified in, for example, the MIPS ISA. In the case of MULT(U), one of the 16-bit Multiply-Accumulate datapaths works iteratively to produce the 64-bit product in five cycles. The least significant 32 bits are available one cycle earlier than the most significant 32 bits. MMD register 500 mode bits have no effect on the operation of the standard MIPS ISA instructions. By contrast, the MULTA instruction is subject to MMD register 500 mode bits for fractional arithmetic and truncated 32x32-bit multiplication.

For the DSP MULTA, an accumulator pair M0h[31:0]/M0l[31:0], Mlh[31:0]/M11[31:0] etc. is the target. M0h[31:0] is aliased to HI; M01[31:0] is aliased to LO. Unlike the (dual) 16-bit operations, single-cycle throughput is not available for 32-bit data. Because there are two

available data paths, however, two 32x32-bit multiply operations can be initiated every four cycles. The Dual MAC hardware configuration consistent with this invention automatically allocates the second operation to the available data path. If a third 32-bit multiplication is programmed too soon, stall cycles are inserted until one of the data paths is free.

The Dual MAC configuration consistent with this invention also supports a complex multiply instruction, CMULTA. For this instruction, each of the 32-bit general register operands is considered to represent a 16-bit real part (in bits 31:16) and a 16-bit imaginary part (in bits 15:00). One of the multipliers calculates the real part (32 bits) of the complex product (namely XrYr - XiYi) and stores it in the "h" half of the target accumulator pair. The other multiplier calculates the imaginary part (32 bits) of the complex product (namely XrYi + XiYr) and stores it in the "*l*" half of the target accumulator pair. This instruction can be initiated every two cycles (2-cycle throughput) and takes four cycles to complete. As in the other Dual MAC operations, programming CMULTA instructions too close together causes stall cycles but the correct results are always obtained.

The Dual MAC configuration consistent with this invention includes a separate Divide Unit 630 for executing the 32-bit DIV(U) operations specified by the MIPS ISA. The Divide requires 19 cycles to complete. The quotient is loaded into M01[31:0], Mll[31:0], M21[31:0], or M31[31:0], and the remainder is loaded into the lower 32-bits of the other accumulator in the target pair. There is no special support for fractional arithmetic for the DIV operations.

Because the Dual MAC configuration consistent with this invention is capable of consuming four 16-bit operands every cycle (in Pipe B) by performing two l6xl6 bit multiply-accumulates, it is desirable to be able to fetch four 16-bit operands from memory every cycle (in

Pipe A). Therefore, the DSP extends the MIPS load and store instructions to include twinword accesses and implements a 64-bit data path from memory. A twinword memory operation accesses an (even-odd) pair of 32-bit general registers with a single instruction and executes in a single pipeline cycle.

Like the standard byte, halfword, and word load/store instructions, the twinword load/store instructions use a register and an immediate field to specify the memory address. However, to fit into the format, the available signed 11-bit immediate field (called the displacement) is considered a twinword quantity, so is left-shifted by 3 bits before being added to the base register. This is equivalent to a 14-bit byte offset, in comparison to the full 16-bit immediate byte offset used in the byte, halfword and word instructions. Also, the target register pair for the twinword load/store must be an even-odd pair, so that only 4 bits are used to specify it.

DSP algorithms usually operate on vectors or matrices of data; for example Discrete Cosine Transforms operate on 8x8 pixel blocks. As a result, data memory pointers are incremented from one operand to the next. The extra instruction cycle required to increment RISC memory pointers is eliminated in DSPs with auto-increment. Memory pointers are used unmodified to create the address, then updated in the general register file before the next use:

address ← pointer

pointer ← pointer + stride

The 8-bit immediate field containing the stride is sign-extended to 32-bits before being added to the pointer for the latter's update. The nomenclature "pointer" distinguishes the update performed after memory addressing, as opposed to the "base" register (in the MIPS ISA), which

is augmented by the offset before addressing memory in the standard instructions. The nomenclature "stride," which depends on the granularity of the access, distinguishes it from the invariant byte offset used in the standard load and store instructions. For twinword/word/ halfword addressing the 8-bit field is first left-shifted by three/two/one places and zero-filled, before sign extension to 32-bits. This use of left shifts for the twinword, word, and halfword strides is similar to the MIPS 16 ASE and is used to extend the effective address range. Thus, increments of between -128 and +127 twinwords, words, halfwords or bytes are available for each data type.

In the case of Loads (but not Stores) pointer update requires a second general register file write port. An 8 port (4read/4write) register file has two of the four write ports dedicated to register Loads. As a result, twinword loads can execute in parallel with any Pipe B operation. Figs. 8A, 8B, 8C, and 8D contain a Table 800 of the instructions supported by an implementation consistent with this invention. Fig. 9 contains a Table 900 showing the assignment of instructions to Pipe A and Pipe B.

5. Circular buffers

For some DSP algorithms, notably filters, DSP data is organized into "circular buffers." In this case, the next reference after the end of the buffer is to the beginning of the buffer. Implementing this structure in RISC requires:

Inst 1:	LW	reg, AddressReg
Inst 2:	BNEL	AddressReg, BufferEnd, Continue
Inst 3:	ADDIU	AddressReg, AddressREG + 4
Inst 4:	MOVE	AddressReg, BufferStart

Continue:

The above example is written so that a branch prediction failure will only be incurred at the end of the buffer. Nevertheless, the combination of post-modified pointers together with hardware support for circular buffers allows reducing this typical DSP addressing operation from four cycles to one.

Fig. 10 is a block diagram illustrating circular buffer start registers (cbs 0 1010, cbs 1 1013, and cbs2 1016) and circular buffer end registers (cbe0 1020, cbe1 1023, and cbe2 1026) preferably located in ALU A 420 (Fig. 4). The circular buffer control logic is quite sophisticated in dealing with different data widths and decrement and well as increment of the circular buffer pointer.

The DSP supports three circular buffers. To initialize the circular buffers, MTALU (Move To ALU) instructions are used to set the twinword start addresses cbs 0 1010, cbs 1 1013, and cbs 2 1016 [31:3] and twinword end addresses cbe 0 1020 cbe 1 1023, and cbe 3 1026 [31:3]. Circular buffers are only used when memory pointers are post-modified, and consist of an integral number of twinwords.

When a circular buffer pointer is used in a post-modified address calculation, the pointer is compared to the associated cbe address. If they match (and the stride is non-negative), the cbs address (rather than the post-modified address) is restored to the register file. Similarly, to allow for traversing the circular buffer in the reverse direction, the pointer is compared to the cbs address; if they match (and the stride is negative) the cbe address (rather than the post-modified address) is restored to the register file.

Also in Fig. 10 are AREG 1030, which holds the memory pointer, BIREG 1035, which holds the "stride" by which the memory pointer is modified, BRREG 1040, which holds the date to be stored in the case of store instructions, and TEMP 1045, which holds a pipeline stage register DADDR_E 1050. The contents of AREG 1030 are transferred to DADDR_E 1050 as the memory address.

A+BI 1060 is the memory pointer modified by the stride; Select signal 1064 indicates whether the memory pointer matches the circular buffer end address; and ALUREG 1070 is the output of the ALU. In this case, ALUREG 1070 holds the modified pointer that will be used in the next such memory address.

Circular buffers can also be accessed with byte, halfword, or word Load/ store with Pointer Increment instructions. In those cases, the several least significant bits of the pointer register are examined to determine if the start or end of the buffer has been reached, taking into account the granularity of the access, before replacing the pointer with the cbs or cbe as appropriate.

Any general register memory pointer can be used with circular buffers using the ".Cn" option. To use general register rP as a circular buffer pointer, for example, the instruction

LWP.C2 r3, (r4)stride

associates the r4 memory pointer with circular buffer C2 defined by the start address cbs 2 1016 and end address cbe 2 1026. Figs. 11A, 11B, and 11C contain a Table 1100 summarizing vector addressing instructions with the implementation described above.

6. Instruction extensions

The DSP accommodates extensions to the standard instruction sets, such as the MIPS instruction set, to support dual 16-bit operations, and also introduces a number of additional ALU instructions that improve performance on DSP algorithms. Supporting high-performance, dual 16-bit operations in the RISC-DSP requires supporting not only dual MAC instructions but also dual 16-bit versions of other arithmetic operations that the programmer may require.

In general, 16-bit immediates are not supported because the DSP extensions are encoded using the INST[05:00] "subop" field. Although dual 16-bit versions of logical operations such as AND may not be required, dual 16-bit versions have been provided for all 3-register operand shifts and add/subtracts. In a preferred implementation, the character "2" in the assembler mnemonic indicates an operation on dual 16-bit data.

DSP algorithms are often somewhat tolerant of data errors. A bad audio sample, for example, may cause a brief distortion, but no lasting effect as new audio samples arrive and the bad sample is cleared out of the buffer. Accordingly, the saturated result of signed arithmetic is a closer, more desirable, approximation than the wraparound result. Therefore, all arithmetic operations that may potentially produce arithmetic overflow or underflow, and do not have immediate operands, support optional saturation. For example, not only the dual 16-bit add (ADDR2), but also the 32-bit add (ADDR) have optional saturate. Neither the dual 16-bit instructions nor the 32-bit saturating adds and subtracts cause exceptions.

A DSP consistent with this invention includes several additional ALU instructions for DSP performance analysis. Consistent with the approach described above, each additional instruction has both a 32-bit and a dual 16-bit version. If signed overflow/underflow is possible,

a saturation option is provided. These instructions are described in Table 1200 in Figs. 12A and 12B.

Some DSPs provide a more extensive set of microcontrol functions; for example, field "set," "clear," and "extract" functions. In ATM cell processing, for example, these functions are useful in processing the cell headers. In DSPs consistent with systems and methods of the present invention, these operations can readily be executed using one or more RISC operations. For example:

```
and     r3, r2, r1 (mask)      // clears a field specified in the mask
or      r3, r2, r1 (mask)      // sets a field specified in the mask
s11     r3, r3, n              // extracts the field between [31 - n : 31 - (n+m)]
srl(sra) r3, r3, m             // the field is right-justified and 0 (sign) extended if
                               // srl (sra) is selected.
```

Figs. 13A and 13B contain Table 1300 that lists and describes additional ALU operations. Several instructions allow conditional execution of Conditional Move (CMV<COND>).

A number of DSPs and RISC processors have deployed extensive conditional execution. In these processors, the branch prediction penalty is three cycles or more. Conditional execution can mitigate the effect of the branch prediction penalty by allowing bypassing of the branch in some cases. Conditional execution is a costly alternative, however, because it uses instruction opcode bits and consequently limits the size of immediates and/or limits the number of general purpose registers visible to the program. The branch prediction penalty in the DSP of the present invention, however, only requires two cycles; therefore the value of conditional execution is minimized and only a restricted set of "conditional move" instructions is needed. The effect of

any conditional execution can be "emulated," however, with a sequence of two instructions by using the conditional move. For example:

Processor with conditional execution:

Inst 1: ALU operation sets condition flags

Inst 2: COND: ALU operation

In the DSP consistent with this invention:

Inst 1: ALU operation updates register rB (condition setting operation)

Inst 2: ALU operation with result directed to temp register rA

Inst 3: CMV<COND> rD, rA, rB

If rB satisfies the COND, rD is updated with rA; i.e. the 2nd ALU operation is executed to "completion." This sequence is interruptible. The if-then-else construct:

```
if (rB COND)
        rD = rA
else
        rD = rC
```

can be coded if the previous example is prefaced with:

```
MOVE rD, rC          // move rC to rD
```

This conditional move facilitates initial porting of assembler code from processors with conditional execution. Table 1400 in Fig. 14 lists conditional operations consistent with this invention.

7. Zero Overhead Loop Facility

DSP algorithms spend much of their time in short real-time critical code loops. To compensate, DSPs often include hardware support for "zero-overhead looping." Zero-overhead looping allows branching from the end-to-beginning of the loop can be accomplished without

explicit program overhead if the loop is to be executed a fixed number of times, known at compile time. Typically, loop counters and program start and/or end address registers are required. Zero-overhead looping cannot be nested without additional hardware. Often, zero-overhead loops cannot be interrupted.

A DSP hardware configuration consistent with the disclosed embodiment supplies such a facility but allows the loop count to be determined at run time as well. The facility consists of three Radiax registers, which are accessible by a program running in User mode using Radiax instructions MFRU and MTRU. The operating system should consider these registers as part of the context of the executing process and should save and restore them in the case of an interrupt. These three registers include:

LPE0[31:2] -- maintaining a virtual address of the ending instruction of the loop;

LPS0[28:2] -- holding low order bits of the virtual address of the "starting" instruction of the loop; and

LPC0[15:0] -- holding the loop count value.

Although this feature is intended to operate in loops, the algorithm executed by the hardware can be described more simply. In particular, it should be noted that there is no knowledge of being "inside" the loop. All that matters is the contents of the three registers when an attempt is made to execute the instruction at the address specified by LPE0:

If (M32-mode, AND current-instruct-addr[31:2] = LPE0, AND LPC0 ≠ 0) then

execute current instruction (at LPE0[31:2] || 00),
decrement LPC0[15:0] by one,
execute instruction at LPE0[31:29] || LPS0[28:2] || 00
continue (LPS0 could be a jump/branch)

Else

execute current instruction,
continue (current instruction could be a jump/branch)

In this embodiment, the order of loading the registers should be LPS, then LPE, then LPC with a non-zero value. Further, there should be at least two (2) instructions between the instruction that loads LPC with a non-zero value, and the instruction at the LPE address. To guarantee that no stall cycles are incurred, at least six (6) instructions should separate the instruction that loads LPC with a non-zero value, and the instruction at the LPE address.

8. Cycle-by-cycle usage for Dual MAC instructions

As explained above, a Dual MAC architecture eliminates programming hazards for its instructions by stalling the pipeline when necessary. It does this both to avoid resource conflicts and to wait for results of a first instruction to be ready before attempting to use those results in a second instruction. This means that no programming restrictions are needed to obtain correct results from a sequence of Dual MAC instructions.

The most efficient use of the hardware, however, occurs when the program avoids these stalls, usually by properly scheduling the instructions. Table 1500 in Fig. 15 lists the cycles required between instructions to assist in such scheduling. Specifically, Table 1500 indicates the number of cycles that must be inserted between the first indicated instruction and the second indicated instruction. The number "0" means that the two instructions can be issued back-to-back. If the instructions are issued back to back, the Dual MAC architecture will stall the pipeline for the indicated number of cycles. Non-Dual MAC instructions can be issued to occupy those cycles, or other appropriate Dual MAC instructions can be issued in those cycles, such as those using non-conflicting accumulators.

The following code sequences indicate the most efficient use of the Dual MAC for coding the inner loop of some common DSP algorithms. The algorithms are presented for 16-bit operands with 16-bit results, as well as 32-bit operands with 32-bit results. The algorithms assume that fractional arithmetic is used. Therefore, for the 32-bit results of a 32x32 multiply, only the HI half of the target accumulator pair is retrieved or used.

In these examples, only the Dual MAC instructions are shown. The other pipe is used to fetch and store operands and take care of loop housekeeping functions. The loops may need to be unrolled to take full advantage of the multiple Dual MAC accumulators.

Case 1: 16-bit inner product SUM = SUM + Ai*Bi

Assuming packed operands, two multiply-adds per cycle:

```
MADDA2    m0,   r1,   r2
MADDA2    m0,   r3,   r4
MADDA2    m0,   r5,   r6
MADDA2    m0,   r7,   r8
...
```

Case 2: 16-bit vector product loop. Ci = Ai*Bi

Assuming packed fractional operands, two multiplies per two cycles using two accumulator pairs.

```
MULTA2    m0,   r1,   r2
MFA2      m1,   r8
MULTA2    m1,   r3,   r4
MFA2      m0,   r7
...
```

Case 3: 16-bit complex vector product. Ci = Ai *complex Bi

Assuming fractional operands packed as 16-bit real, 16-bit imaginary. One complex multiply every two cycles using two accumulator pairs.

```
CMULTA    m0,   r1,   r2
MFA2      m1,   r8
CMULTA    m1,   r3,   r4
MFA2      m0,   r7
...
```

Case 4: 32-bit inner product loop: SUM = SUM + Ai*Bi

Assuming a multiply-add every other cycle using one accumulator.

```
MADDA          m0,   r1,   r2
non-DualMAC op
MADDA          m1,   r3,   r4
non-DualMAC op
```

Case 5: 32-bit vector product loop. Ci = Ai*Bi

Assuming fractional 32-bit operands so that the MFA waits for the HI result of the MULTA. Achieves one multiply per two cycles using all the accumulators.

```
MULTA     m0,   r1,   r2
MFA       r9,   m1h
MULTA     m1,   r3,   r4
MFA       r10,  m2h
MULTA     m2,   r5,   r6
MFA       r11,  m3h
MULTA     m3,   r7,   r8
MFA       r12,  m0h
```

Case 6: 32-bit complex vector product. Ci = Ai *complex Bi

Assuming fractional 32-bit operands so that the ADDMA/SUBMA waits for the HI result of the second MULTA. Achieves one complex multiply per ten cycles using all the accumulators, with two inserted instructions. This is a good example of the cycles needed from MULTA to SUBMA/ADDMA (5 cycles for HI) and from SUBMA/ADDMA to MFA (2 cycles).

MULTA	m0, r1, r4	; m[2i]	*	b[2i+1]
MFA	rigmag, a1h			
MULTA	m1, r2, r3	; m[2i+1]	*	b[2i]
SUBMA	m3h, m2h, a3h	;		c[2i-2] = m[2i-2] *b[2i-2] - m[2i-1] *b[2i-1]
non-DualMac	op			
MULTA	m2, r1, r3	; m[2i]	*	b[2i]
MFA	rreal, m3h			
MULTA	m3, r2, r4	; m[2i+1]	*	b[2i+1]
ADDMA	m1h, m0h, m1h	;		c[2i+1] = a[2i+1] *b[2i] + a[2i] *b[2i+1]

9. Low-overhead interrupts

The DSP consistent with this invention accommodates eight low-overhead hardware interrupt signals that are useful for real-time applications. These Interrupts are supported with three coprocessor 0 registers: ESTATUS (0) 1610, ECAUSE (0) 1620, and INTVEC (0) 1630, which are illustrated in Fig. 16.

ESTATUS register 1610 contains the mask bits IM[15-8] for the low priority interrupt signals. IM[15-8] is reset to 0 so that, regardless of the global interrupt enable, IEc, none of the interrupts will be activated. IP[15-8] for the interrupt signals is located in ECAUSE 1620. Each of these fields are similar to IM and IP defined in the R3000 Exception Processing model. One

difference is that the interrupts are prioritized in hardware and each have a dedicated Exception Vector.

IP[15] has the highest interrupt priority, IP[14] next, etc. All new interrupts are higher priority than IP[0-7]. The Exception vector for the interrupts is located in INTVEC register1630. The BASE for these vectors is program-defined. The vector for IP[15] is BASE || 111000 ||, for IP[14] BASE || 110000 ||, etc. Thirty-two instructions can be executed at each vector; if more are required the program can jump to any address.

10. Operational Codes

DSP architecture consistent with this invention allows for an extension of a standard instruction set by designating a single I-Format as "LEXOP," then using the INST[5:0] "subop" field to permit up to 64 additional opcodes. Thus, the additional DSP opcodes model the MIPS "special" opcodes encoded in R-Format. The diagrams in Figs. 17A-H illustrate an example of the type of opcode extensions the present invention can designate. In this example, a "LEXOP" designation code with an I-Format 011_111 is used. Additional DSP opcodes can be integrated into an established MIPS instruction set by using a relocatable code to designate 64 subops.

In the present embodiment, the default object code for LEXOP is 011_111. However, the location can be moved using power/ground straps, for example. These straps or similar configurable devices help insure compatibility of the DSP extensions with future MIPS ISA extensions. The code assigned to "LEXOP" can be moved around with external power/ground straps to insure long-term compatibility of the DSP extensions with the MIPS ISA.

The following principles are used to resolve potential ambiguity of encoding between the DSP of the present invention extensions and instructions:

a. Instructions with similar operations to an existing MIPS instruction set, but with additional operands permitted, are programmed with additional Assembler mnemonics and encoded as a LEXOP. For instance:

multa	ml, rl, r2	is encoded as a LEXOP instruction
mult	rl, r2	is encoded as a MIPS instruction
multa	m0, rl, r2	is encoded as a LEXOP instruction (a0 is an alias for HI/LO).

b. If a MIPS instruction is "extended" with additional functionality, it is programmed with additional Assembler mnemonics and encoded as a LEXOP. In the disclosed example, mnemonics ending in "r" indicate general register file targets; mnemonics ending in "m" indicate accumulator register targets. This convention removes ambiguity between the Lexra op and a similar MIPS op. For example,

addr	r3, rl, r2	is encoded as a LEXOP instruction
add	r3, rl, r2	is encoded as a MIPS instruction

The MIPS *add* and the LEXOP *addr* are both signed 32-bit additions. However, on overflow the MIPS instruction triggers the Overflow Exception, while the LEXOP does not. Alternatively, the result of the LEXOP will saturate if the ".s" option is selected (addr.s).

The foregoing description is presented for purposes of illustration and description. It is not exhaustive and does not limit the invention to the precise form disclosed. Modifications and variations are possible in light of the above teachings or may be acquired from practicing the invention. The scope of the invention is defined by the claims and their equivalents.

WHAT IS CLAIMED IS:

1. A digital signal processor comprising:

two execution pipelines capable of executing RISC instructions;

instruction fetch logic that simultaneously fetches two instructions and routes them to respective pipelines; and

control logic to allow the pipelines to operate independently.

2. The digital signal processor of claim 1, wherein the instruction fetch logic includes logic that fetches dual SIMD instructions.

3. The digital signal processor of claim 1, further including two registers each half the length of a word fetched for memory, and

wherein the instruction fetch logic that fetches a single word into the two registers simultaneously.

4. The digital signal processor of claim 1, further including an eight port general register file.

5. The digital signal processor of claim 4, wherein the general register file includes four read registers and

four write registers.

6. A digital signal processor capable of integrating subopcodes into an established instruction set comprising:

a memory that stores instructions having opcodes;

an instruction decoder that identifies a relocatable opcode to designate 64 subopcodes; and

a subopcode detector that decodes subopcodes if the instruction decoder identifies the relocatable opcode.

7. A digital signal processor comprising:

a register pair; and

means for executing a multiply instruction on a number stored in the register pair, including

first means for performing multiply instructions on higher-order portions of each register in the register pair,

second means for performing multiply instructions on the remaining portions of each register in the register pair, and

third means for combining the results from the first and second means.

8. A circular buffer control circuit comprising:

a first number of circular buffer start registers;

a first number of circular buffer end registers, each associated with a different one of the circular buffer start registers; and

circular buffer control logic including

means for comparing a pointer to an address in a selected one of the circular buffer end registers, and

means for restoring the address in the one of the circular buffer start registers associated with the selected circular buffer end register if the pointer matches the address in the selected circular buffer end register.

9. A digital signal processor capable of executing zero overhead looping instruction commands comprising:

a register set; and

means for executing a loop instruction command a fixed number of times on a number stored in the register set, including

first means for executing a current instruction stored in a first portion of a first register within the register set;

second means for decrementing a loop count value stored in a second register within the register set;

third means for executing another portion of the current instruction stored in a second portion of the first register and a second register within the register set.

10. The digital signal processor of claim 9, further including

means for exiting the loop instruction command when the loop count value reaches zero.

ABSTRACT

A DSP superscalar architecture employing dual multiply accumulate pipelines. Dual MAC pipelines allow for a seem less transition between established RISC instruction sets and extended DSP instructions sets. Relocatable opcodes are provide to allow further extensions of RISC instruction sets. The DSP superscalar architecture also provides memory pointers with hardware circular buffer support, an interruptible and nested zero-overhead loop counter, and prioritized low-overhead interrupts.

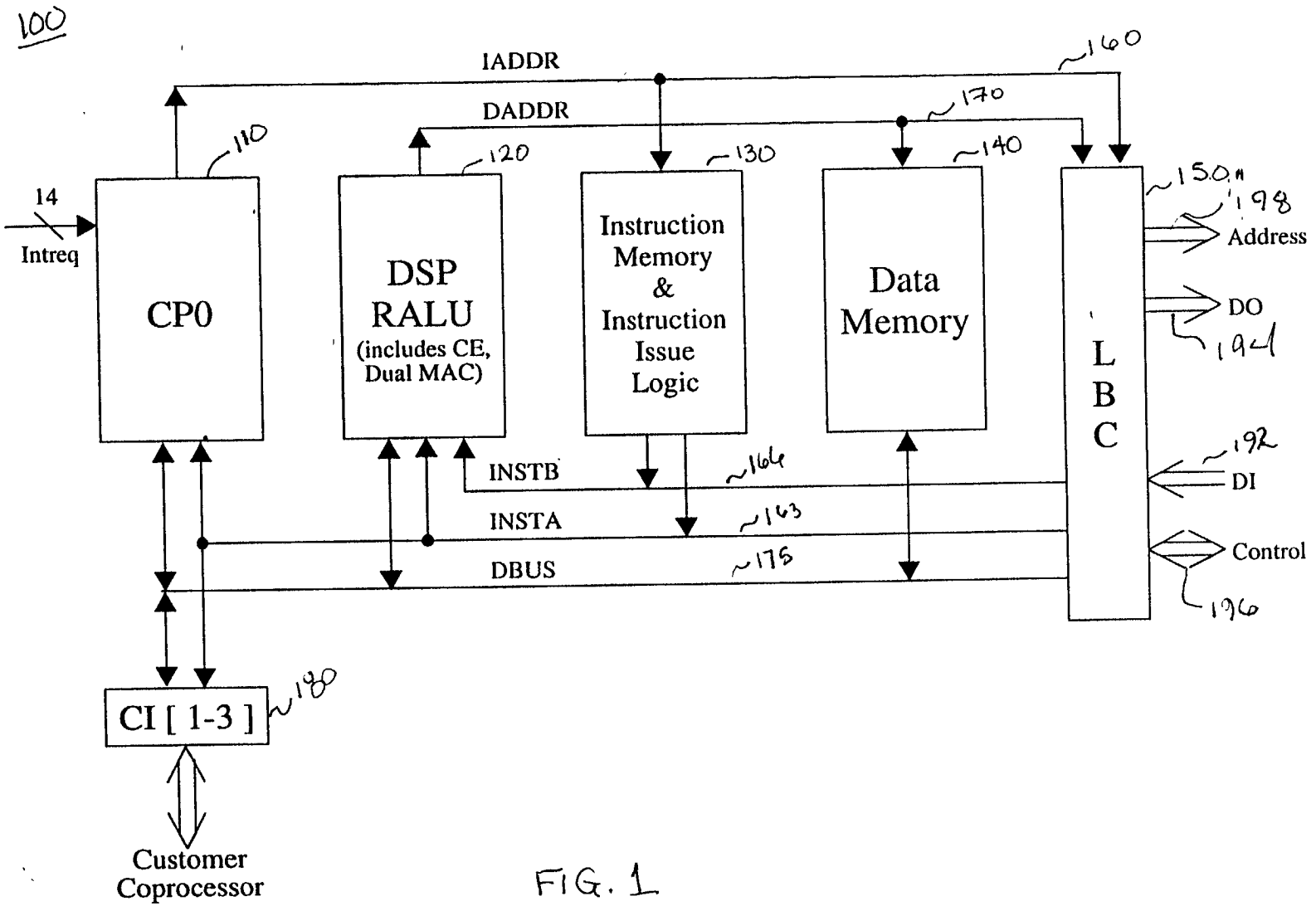

100
IADDR
160
DADDR
170
110
120
130
140
150
14
Intreq
CP0
DSP
RALU
(includes CE,
Dual MAC)
Instruction
Memory
&
Instruction
Issue
Logic
Data
Memory
L
B
C
198
Address
DO
194
192
DI
INSTB
164
INSTA
163
DBUS
175
Control
196
CI [1-3]
180
Customer
Coprocessor

FIG. 1

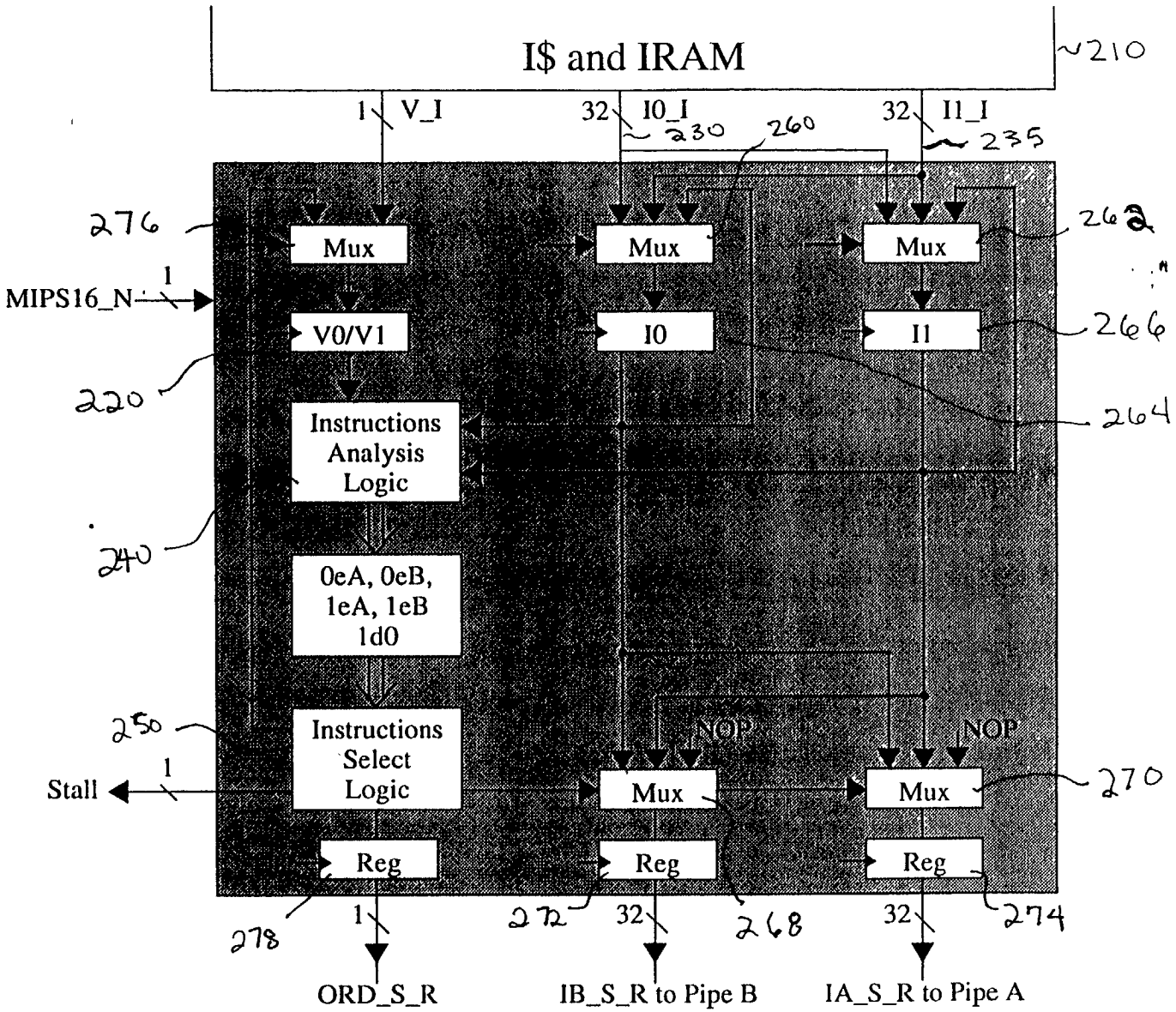


FIG 2

: **Instruction Select Logic**

300

		0eA : 0eB			
		00 (I0 not valid)	**01**	**11**	**10**
1eA : 1eB	00 (I1 not valid)	IA←NOP IB←NOP ORD←d/c V[1:0]←next	Not possible	Not possible	Not possible
	01	IA←NOP IB←I1 ORD←B V[1:0]←next	IA←NOP IB←I0 Stall←1 ORD←B V[1:0]←01	IA←I0 IB←I1 ORD←A V[1:0]←next	IA←I0 IB←I1 ORD←A V[1:0]←next
	11	IA←NOP IB←I1 ORD←B V[1:0]←next	IA←I1 IB←I0 ORD←B V[1:0]←next	IA←I0 IB←I1 ORD←A V[1:0]←next	IA←I0 IB←I1 ORD←A V[1:0]←next
	10	IA←I1 IB← NOP ORD←A V[1:0]←next	IA←I1 IB←I0 ORD←B V[1:0]←next	IA←I1 IB←I0 ORD←B V[1:0]←next	IA←I0 IB←NOP Stall←1 ORD←A V[1:0]←01
1d0		Not possible	IA←NOP IB←I0 Stall←1 ORD←B V[1:0]←01	IA←I0 IB←NOP Stall←1 ORD←A V[1:0]←01	IA←I0 IB←NOP Stall←1 ORD←A V[1:0]←01

* Note: I0 valid and I1 not valid won't occur because there is no out of order execution.

FIG.3

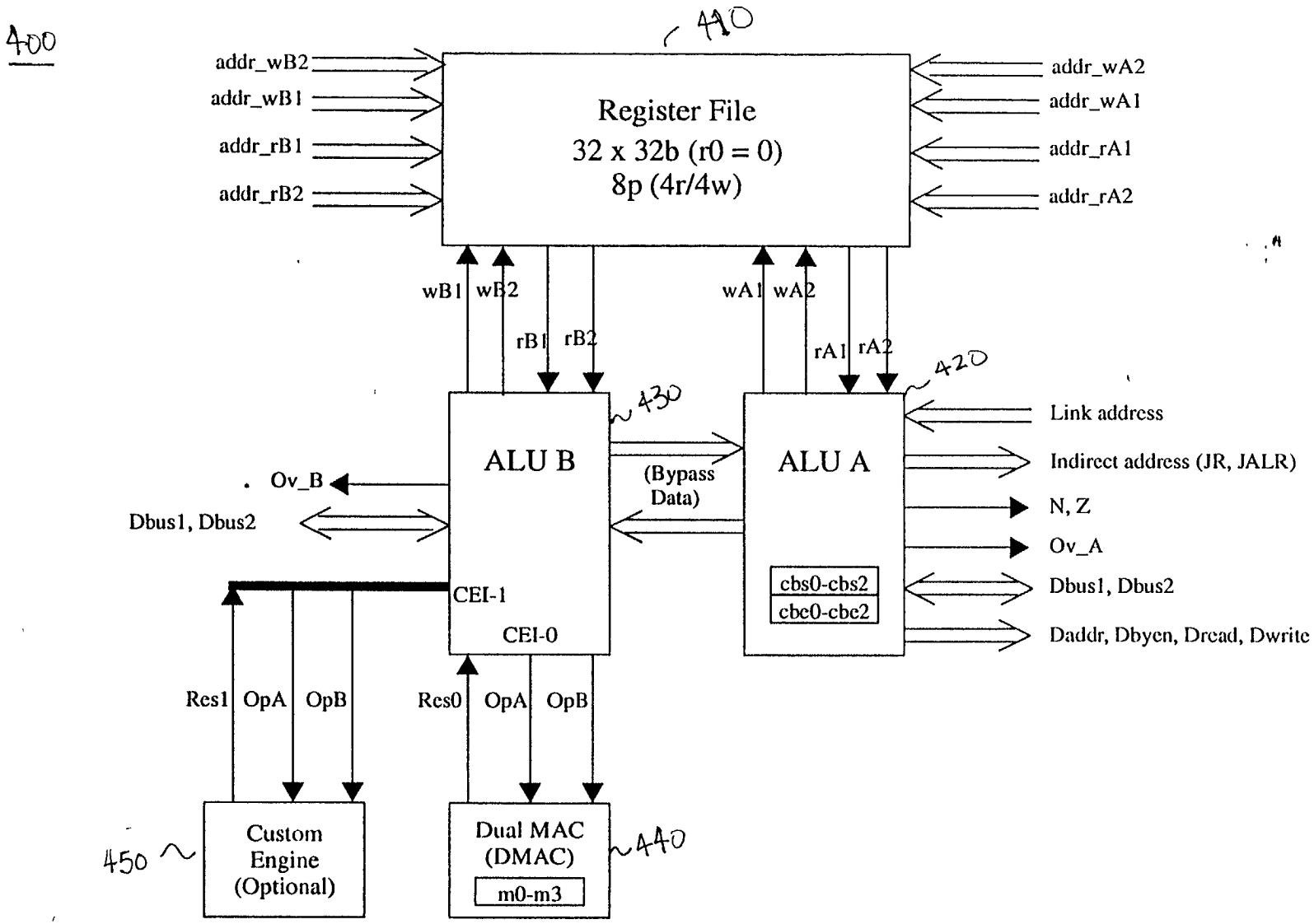
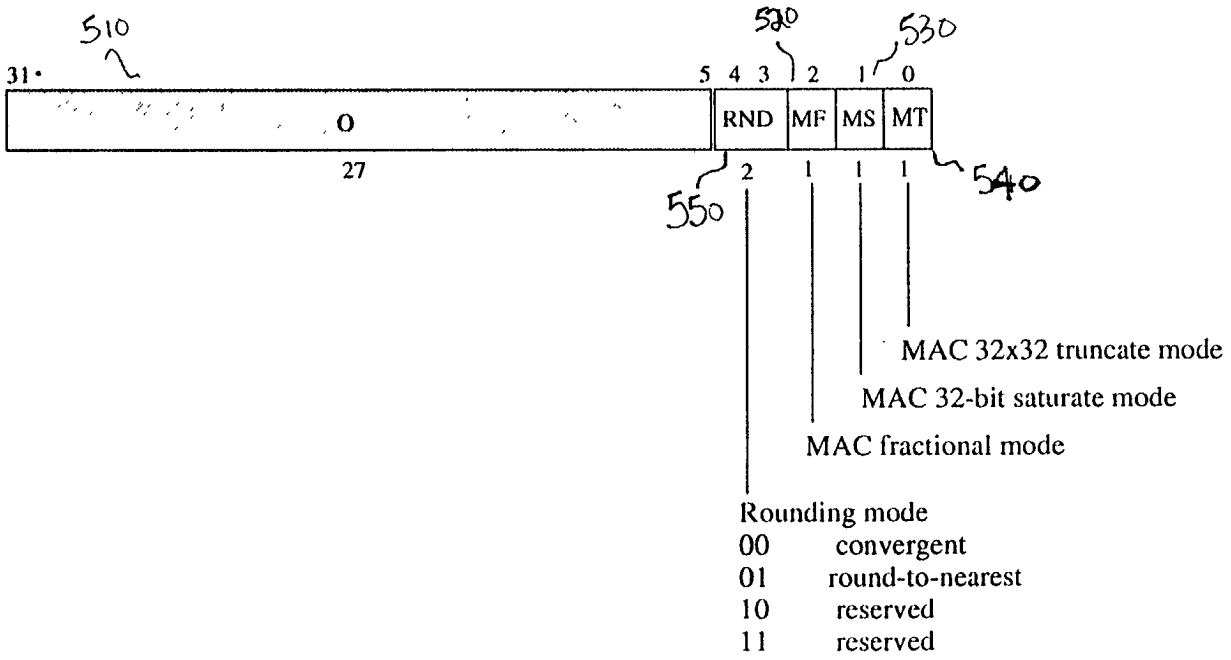


FIG. 4

MMD (Radiax User Register 24)



500

FIG. 5

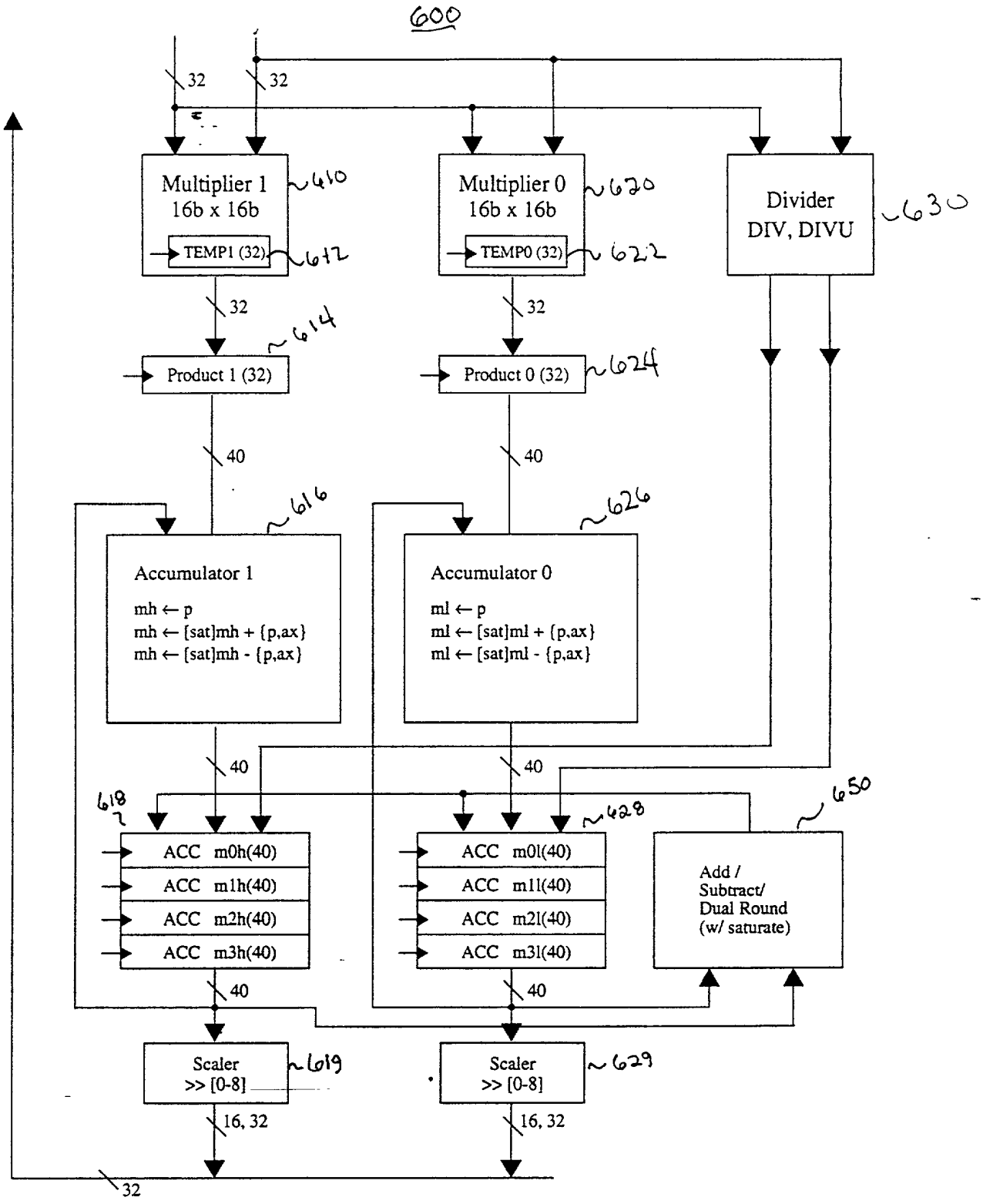

600
32
32
Multiplier 1
16b x 16b
610
TEMP1 (32)
612
Multiplier 0
16b x 16b
620
TEMP0 (32)
622
Divider
DIV, DIVU
630
32
32
614
Product 1 (32)
Product 0 (32)
624
40
40
616
626
Accumulator 1
mh ← p
mh ← [sat]mh + {p,ax}
mh ← [sat]mh - {p,ax}
Accumulator 0
ml ← p
ml ← [sat]ml + {p,ax}
ml ← [sat]ml - {p,ax}
40
40
618
628
650
ACC m0h(40)
ACC m1h(40)
ACC m2h(40)
ACC m3h(40)
ACC m0l(40)
ACC m1l(40)
ACC m2l(40)
ACC m3l(40)
Add /
Subtract/
Dual Round
(w/ saturate)
40
40
Scaler
>> [0-8]
619
Scaler
>> [0-8]
629
16, 32
16, 32
32

FIG. 6

- Integer format interpretation:

X =

S	14	13	12			- - -	0

^
(radix point)

$$X = -2^{15} \times S + \sum_{0}^{14} X[i]2^{i}$$

Product = P[31:0] is sign-extended to P[39:0] = $P[31]^8$ || P[31:0] for accumulation

P =

S	30	29	28			- - -	0

^
(radix point)

$$P = -2^{31} \times S + \sum_{0}^{30} X[i]2^{i}$$

- Fractional format interpretation:

X =

S	14	13	12			- - -	0

^
(radix point)

$$X = -1 \times S + \sum_{0}^{14} X[i]2^{i-15}$$

Product = P[31:0] is left-shifted one bit and sign-extended to:
P[39:0] = $P[30]^8$ || P[30:0] || 0 for accumulation

P =

S	29	28	27			0	zero

^
(radix point and product left shifted by one)

$$P = -1 \times S + \sum_{0}^{29} P[i]2^{i-30}$$

FIG. 7A

Overflow Protection: Guard Bits and Saturation

- The LX5280 accumulator implements eight (8) guard bits to protect against overflow. The alternatives of (i) product scaling or (ii) input scaling by right shifting, cause loss of precision.

- Optional saturation (MADDA2.S, MSUBA2.S, ADDMA.S, SUBMA.S) can be used to avoid wrap-around on underflow or overflow of the 40-bit format (or 32-bits if that mode is selected in the MMD register):

  if (result > 0 1 1 1 1 ... 1) result = 0 1 1 1 1 ... 1

  if (result < 1 0 0 0 0 ... 0) result = 1 0 0 0 0 ... 0

- In 32-bit saturate mode, the MAC implements a full 40-bit saturation detector. This allows for the case where the accumulator holds a value greater than the maximum 32-bit saturated value prior to the addition (or subtraction) with saturation.

FIG. 7B

MAC Output Control: Rounding and Scaling

- For output storage, the 40-bit accumulators must be converted to 16-bit or 32-bit format.
- Scaling

  single accumulator:

  $$RES[31:0] \leftarrow \{ mTh, mTl \}\ [31 + n : n] \qquad [n = 0 - 8]$$

  dual accumulators (select high half of each, useful for fractional arithmetic results):

  $$RES[31:0] \leftarrow mTh\ [31 + n : 16 + n]\ \|\ mTl\ [31 + n : 16 + n]\ [n = 0 - 8]$$

- To avoid the bias introduced by truncation, the accumulator can be rounded prior to output scaling (useful for fractional arithmetic results).

  $$\{ mT, mTh, mTl \} \leftarrow RNDA2\ \{ mT, mTh, mTl \} \qquad [n = 0 - 8]$$

  $\Rightarrow$ the rounding mode is selectable in the MMD register

  bit position 16 + n of each accumulator in the pair is the least significant bit of precision after rounding

FIG. 7C

MAC Radiax Instruction Summary

800

Instruction	Syntax and Description
Dual Move to Accumulator	*MTA2[.G] rS, {mD, mDh, mDl}* If MTA2, and mDh(mDl) is selected, sign-extend the contents of general register rS to 40-bits and move to accumulator register mDh(mDl). If MTA2, and mD is selected, update both mDh and mDl with the 40-bit, sign-extended contents of the same rS. If MTA2.G is selected, the accumulator register bits [39:32] are updated with rS[31:24]; bits [31:00] of the accumulator are unchanged. (The .G option is used to restore the upper-bits of the accumulator from the general register file; typically, following an Exception.)
Move From Accumulator	*MFA rD, {mTh, mTl} [,n]* Move the contents of accumulator register mTh or accumulator register mTl to register rD with optional right shift. Bits [31+n : n] from the accumulator register are transferred to rD[31:00]. The range n = 0 - 8 is permitted for the output alignment shift amount. In the case of n = 0, the field may be omitted.
Dual Move From Accumulator	*MFA2 rD, mT [,n]* Move the contents of the upper halves of accumulator register pair mT to register rD with optional right shift. The rD[31:16] are taken from mTh and rD[15:00] from the corresponding mTl. mTh[31+n: 16+n] || mTl[31+n : 16+n] from the accumulator register pair are transferred to rD[31:00]. The range n = 0 - 8 is permitted for the output alignment shift amount. In the case of n = 0, the field may be omitted.
Divide	*DIVA mD, rS, rT* The contents of register rS is divided by rT, treating the operands as signed 2's complement values. The remainder is sign-extended to 40-bits and stored in mDh and the quotient is sign-extended to 40-bits and stored in mDl. m0h[31:00] is also called HI. m0l[31:00] is also called LO.
Divide Unsigned	*DIVAU mD, rS, rT* The contents of register rS is divided by rT, treating the operands as unsigned values. The remainder is zero-extended to 40-bits and stored in mDh and the quotient is zero-extended to 40-bits and stored in mDl. m0h[31:00] is also called HI. m0l[31:00] is also called LO.
Multiply (32-bit)	*MULTA mD, rS, rT* The contents of register rS is multiplied by rT, treating the operands as signed 2's complement values. The upper 32-bits of the 64-bit product is sign-extended to 40-bits and stored in mDh and the lower 32-bits is zero-extended to 40-bits and stored in the corresponding mDl. m0h[31:00] is also called HI. m0l[31:00] is also called LO. If MMD[MT] is 1, then the partial product rS[15:00] x rT[15:00] is not included in the total product. If MMD[MF] is 1, then the product is left shifted by one bit, and furthermore, if both operands are -1 then the product is set to positive signed, all ones fraction, prior to the shift. If both MMD[MT] and MMD[MF] are 1, the result is undefined.

FIG. 8A

Instruction	Syntax and Description
Multiply Unsigned (32-bit)	*MULTAU mD, rS, rT* The contents of register rS is multiplied by rT, treating the operands as unsigned values. The upper 32-bits of the 64-bit product is zero-extended to 40-bits and stored in mDh and the lower 32-bits is zero-extended to 40-bits and stored in the corresponding mDl. m0h[31:00] is also called HI. m0l[31:00] is also called LO. If MMD[MT] is 1, then the partial product rS[15:00] x rT[15:00] is not included in the total product. If MMD[MF] is 1, then the result is undefined.
Dual Multiply (16-bit)	*MULTA2 {mD, mDh, mDl}, rS, rT* The contents of register rS is multiplied by rT, treating the operands as signed 2's complement values. If the destination register is mDh, rS[31:16] is multiplied by rT[31:16] and the product is sign-extended to 40-bits and stored in mDh. If the destination register is mDl, rS[15:00] is multiplied by rT[15:00] and the product is sign-extended to 40-bits and stored in mDl. If the destination is mD, both operations are performed and the two products are stored in the accumulator register pair mD. If MMD[MF] is 1, then each product is left shifted by one bit, and furthermore, for each multiply, if both operands are -1 then the product is set to positive signed, all ones fraction.
Dual Multiply and Negate (16-bit)	*MULNA2 {mD, mDh, mDl}, rS, rT* The contents of register rS is multiplied by rT, treating the operands as signed 2's complement values. If the destination register is mDh, rS[31:16] is multiplied by rT[31:16] and the product is sign-extended to 40-bits, negated (i.e. subtracted from zero) and stored in mDh. If the destination register is mDl, rS[15:00] is multiplied by rT[15:00] and the product is sign-extended to 40-bits, negated (i.e. subtracted from zero) and stored in mDl. If the destination is mD, both operations are performed and the two products are stored in the accumulator register pair mD. If MMD[MF] is 1, then each product is left shifted by one bit prior to sign-extension and negation, and furthermore, for each multiply, if both operands are -1 then the product is set to positive signed, all ones fraction prior to sign-extension and negation.
Complex Multiply,	*CMULTA mD, rS, rT* rS[31:16] is interpreted as the real part of a complex number. rS[15:00] is interpreted as the imaginary part of the same complex number. Similarly for the contents of general register rT. As the result of CMULTA, mDh is updated with the real part of the product, sign-extended to 40-bits and mDl is updated with the imaginary part of the product, sign-extended to 40-bits. If MMD[MF] is 1, then each product is left shifted by one bit, and furthermore, for each multiply, if both operands are -1 then the product is set to positive signed, all ones fraction, prior to the addition of terms.

800

FIG. 8B

Instruction	Syntax and Description
32-bit Multiply-Add with 72-bit accumulate	*MADDA mD, rS, rT* The contents of register rS is multiplied by rT treating the operands as signed 2's complement values. If MMD[MT] is 1, then the partial product rS[15:00] x rT[15:00] is not included in the total product. If MMD[MF] is 1, then the product is left shifted by one bit, and furthermore, if both operands are -1 then the product is set to a positive signed, all ones fraction. If both MMD[MT] and MMD[MF] are 1, then the result of the multiply is undefined. The 64-bit product is sign-extended to 72-bits and added to the concatenation mDh[39:0] || mDl[31:0], ignoring mDl[39:32]. The lower 32 bits of the result are zero-extended to 40-bits and stored into mDl. The upper 40-bits of the result are stored into mDh.
32-bit unsigned Multiply-Add with 72-bit accumulate	*MADDAU mD, rS, rT* The contents of register rS is multiplied by rT treating the operands as unsigned values. If MMD[MT] is 1, then the partial product rS[15:00] x rT[15:00] is not included in the total product. If MMD[MF] is 1, then the result of the multiply is undefined. The 64-bit product is zero-extended to 72-bits and added to the concatenation mDh[39:0] || mDl[31:0], ignoring mDl[39:32]. The lower 32 bits of the result are zero-extended to 40-bits and stored into mDl. The upper 40-bits of the result are stored into mDh.
Dual Multiply-Add, optional saturation	*MADDA2[.S] {mD, mDh, mDl}, rS, rT* The contents of register rS is multiplied by rT and added to an accumulator register, treating the operands as signed 2's complement values. If the destination register is mDh, rS[31:16] is multiplied by rT[31:16] then sign-extended and added to mDh[39:00]. If the destination register is mDl, rS[15:00] is multiplied by rT[15:00] then sign-extended and added to mDl[39:00]. If the destination is mD, both operations are performed and the two results are stored in the accumulator register pair mD. If MADDA2.S the result of each addition is saturated before storage in the accumulator register. The multiplies are subject to MMD[MF] as in MULTA2. The saturation point is selected as either 40 or 32 bits by MMD[MS].
32-bit Multiply-Subtract with 72-bit accumulate	*MSUBA mD, rS, rT* The contents of register rS is multiplied by rT treating the operands as signed 2's complement values. If MMD[MT] is 1, then the partial product rS[15:00] x rT[15:00] is not included in the total product. If MMD[MF] is 1, then the product is left shifted by one bit, and furthermore, if both operands are -1 then the product is set to a positive signed, all ones fraction. If both MMD[MT] and MMD[MF] are 1, then the result of the multiply is undefined. The 64-bit product is sign-extended to 72-bits and subtracted from the concatenation mDh[39:0] || mDl[31:0], ignoring mDl[39:32]. The lower 32 bits of the result are zero-extended to 40-bits and stored into mDl. The upper 40-bits of the result are stored into mDh.

FIG. 8C

Instruction	Syntax and Description
32-bit unsigned Multiply-Subtract with 72-bit accumulate	*MSUBAU mD, rS, rT* The contents of register rS is multiplied by rT treating the operands as unsigned values. If MMD[MT] is 1, then the partial product rS[15:00] x rT[15:00] is not included in the total product. If MMD[MF] is 1, then the result of the multiply is undefined. The 64-bit product is zero-extended to 72-bits and subtracted from the concatenation mDh[39:0] || mDl[31:0], ignoring mDl[39:32]. The lower 32 bits of the result are zero-extended to 40-bits and stored into mDl. The upper 40-bits of the result are stored into mDh.
Dual Multiply-Sub, optional saturation	*MSUBA2[.S] {mD, mDh, mDl}, rS, rT* The contents of register rS is multiplied by rT and subtracted from an accumulator register, treating the operands as signed 2's complement values. If the destination register is mDh, rS[31:16] is multiplied by rT[31:16] then sign-extended and subtracted from mDh[39:00]. If the destination register is mDl, rS[15:00] is multiplied by rT[15:00] then sign-extended and subtracted from mDl[39:00]. If the destination is mD, both operations are performed and both results are stored in the accumulator register pair mD. If MSUBA2.S the result of each subtraction is saturated before storage in the accumulator register.
Add Accumulators	*ADDMA[.S] mD{h,l}, mS{h,l}, mT{h,l}* The contents of accumulator mTh or mTl is added to the contents of accumulator mSh or mSl, treating both registers as signed 40-bit values. mDh or mDl is updated with the result. If ADDMA.S, the result is saturated before storage. The saturation point is selected as either 40 or 32 bits by MMD[MS].
Subtract Accumulators	*SUBMA[.S] mD{h,l}, mS{h,l}, mT{h,l}* The contents of accumulator mTh or mTl is subtracted from the contents of accumulator mSh or mSl, treating both registers as signed 40-bit values. mDh or mDl is updated with the result. If SUBMA.S, the result is saturated before storage.The saturation point is selected as either 40 or 32 bits by MMD[MS].
Dual Round	*RNDA2 {mT, mTh, mTl} [,n]* The accumulator register mTh or mTl is rounded, then updated. If mT, the accumulator register pair mTh/mTl are each rounded, then updated. The rounding mode is selected in MMD field "RND". The least significant bit of precision in the accumulator register after rounding is: 16+n. Bits [15+n : 00] are zeroed. The range n = 0 - 8 is permitted for the output alignment shift amount. In the case of n = 0, the field may be omitted.

Nomenclature:

rS, rT	=	r0 - r31
mD	=	mDh || mDl; also for mT
mDh	=	m0h - m3h; also for mSh, mTh
mDl	=	m0l - m3l; also for mSh, mTh
HI	=	m0h[31:00]
LO	=	m0l[31:00]

FIG. 8D

Assignment of Instructions of Pipe A, Pipe B

900

	Pipe A	Pipe B
	The Load/Store Pipe	**The MAC Pipe**
MIPS 32-bit General Instructions	MIPS 32-bit General Instructions except: CE1 Custom Engine Opcodes, MULT(U), DIV(U), MFHI, MFLO, MTHI, MTLO,MAD(U),MSUB(U)	MULT(U), DIV(U), MFHI, MFLO, MTHI, MTLO,MAD(U),MSUB(U) CE1 Custom Engine Opcodes, MIPS 32-bit ALU Instructions Note: No Load or Store Instructions
MIPS 32-bit Control Instructions	J, JAL, JR, JALR, JALX SYSCALL, BREAK, All Branch Instructions, All COPz, SWCz, LWCz	
MIPS16 Instructions (No Doubleword Instructions)	All MIPS16 Instructions except: MULT(U), DIV(U), MFHI, MFLO	MULT(U), DIV(U), MFHI, MFLO
EJTAG Instructions	DERET, SDBBP (including MIPS16 SDBBP)	
Lexra Control Instructions	MTRU, MFRU, MTRK, MFRK, MTLXC0,MFLXC0	
Lexra Vector Addressing	LT, ST, LTP, LWP, LHP(U), LBP(U), STP, SWP, SHP, SBP	
Lexra MAC Instructions		MTA2, MFA, MFA2, MULTA, MULTA2, MULNA2, CMULTA, MADDA, MSUBA, ADDMA, SUBMA, DIVA, RNDA2
Lexra Extensions to MIPS ALU Instructions	SLLV2, SRLV2, SRAV2, ADDR, ADDR2, SUBR, SUBR2, SLTR2	SLLV2, SRLV2, SRAV2, ADDR, ADDR2, SUBR, SUBR2, SLTR2
New Lexra ALU Operations	MIN, MIN2, MAX, MAX2, ABSR, ABS2, CLS, MUX2, BITREV, CMVEQZ, CMVNEZ	MIN, MIN2, MAX, MAX2, ABSR, ABS2, CLS, MUX2, BITREV, CMVEQZ, CMVNEZ

FIG. 9

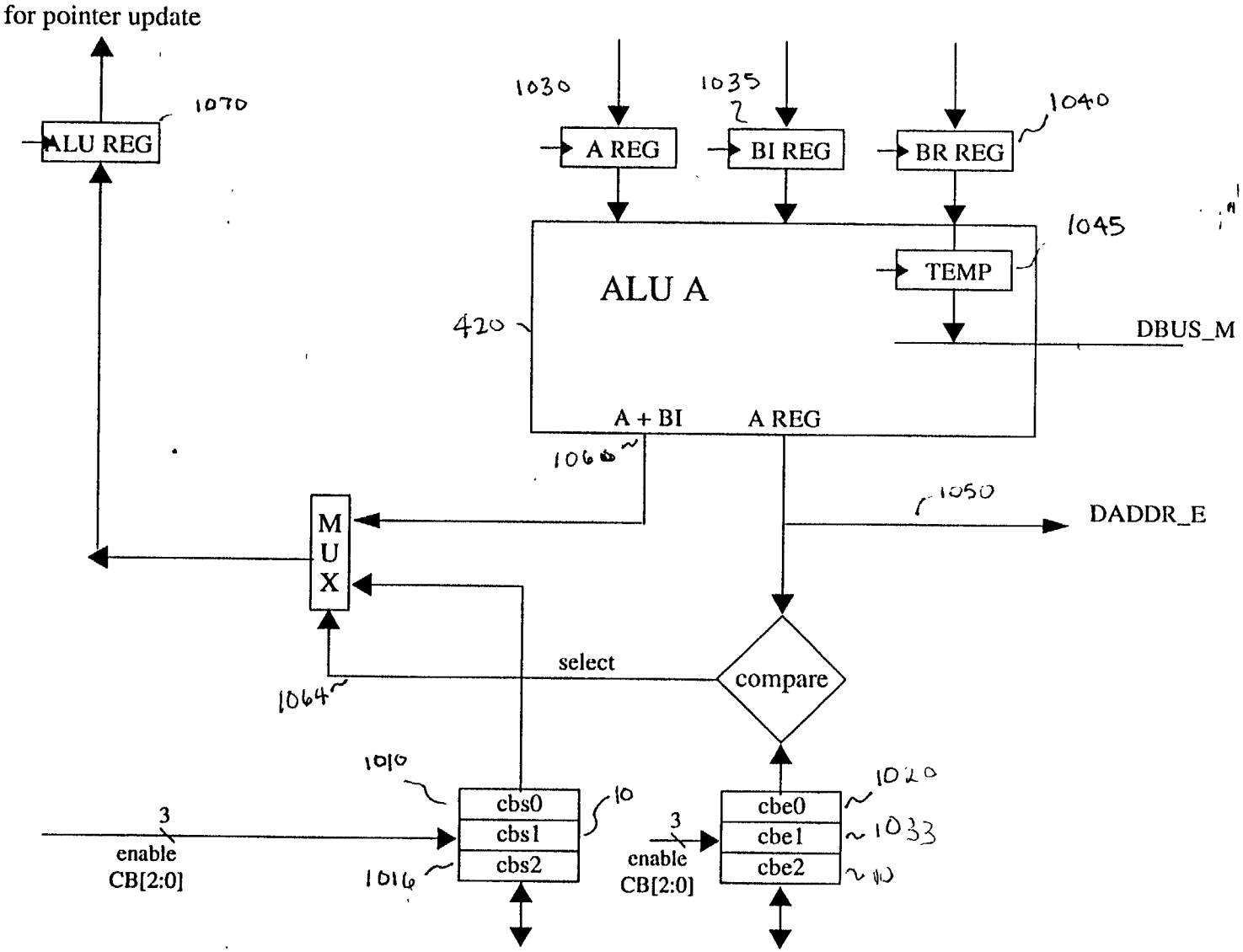


FIG. 10

Vector Addressing Instruction Summary

1100

Instruction	Syntax and Description
Load Twinword	*LT rT, displacement(base)* The displacement, in bytes, is a signed 14-bit quantity that must be divisible by 8 (since it occupies only 11 bits of the instruction word). Sign-extend the displacement to 32-bits and add to the contents of register *base* to form the address *temp*. Load contents of word addressed by *temp* into register rT (which must be an even register). Load contents of word addressed by *temp*+4 into register rT+1.
Store Twinword	*ST rT, displacement(base)* The displacement, in bytes, is a signed 14-bit quantity that must be divisible by 8 (since it occupies only 11 bits of the instruction word). Sign-extend the displacement to 32-bits and add to the contents of register *base* to form the address *temp*. Store contents of register rT (which must be an even register) into word addressed by *temp*. Store contents of register rT+1 into word addressed by *temp*+4.
Load Twinword, Pointer Increment, optional circular buffer	*LTP[.Cn] rT, (pointer)stride* Let *temp* = contents of register *pointer*. Load contents of word addressed by *temp* into register rT (which must be an even register). Load contents of word addressed by *temp*+4 into register rT+1. The stride, in bytes, is a signed 11-bit quantity that must be divisible by 8 (since it occupies only 8 bits of the instruction word). Sign-extend the stride to 32-bits and add to contents of register *pointer* to form next address. Update *pointer* with the calculated next address. ".Cn" selects circular buffer n = 0 - 2. See Note 2.
Load Word, Pointer Increment, optional circular buffer	*LWP[.Cn] rT, (pointer)stride* Load contents of word addressed by register *pointer* into register rT. The stride, in bytes, is a signed 10-bit quantity that must be divisible by 4 (since it occupies only 8 bits of the instruction word). Sign-extend the stride to 32-bits and add to contents of register *pointer* to form next address.Update *pointer* with the calculated next address. ".Cn" selects circular buffer n = 0 - 2. See Note 2.
Load Halfword, Pointer Increment, optional circular buffer	*LHP[.Cn] rT, (pointer)stride* Load contents of sign-extended halfword addressed by register *pointer* into register rT. The stride, in bytes, is a signed 9-bit quantity that must be divisible by 2 (since it occupies only 8 bits of the instruction word). Sign-extend the stride to 32-bits and add to contents of register *pointer* to form next address. Update *pointer* with the calculated next address. ".Cn" selects circular buffer n = 0 - 2. See Note 2.
Load Halfword Unsigned, Pointer Increment, optional circular buffer	*LHPU[.Cn] rT, (pointer)stride* Load contents of zero-extended halfword addressed by register *pointer* into register rT. The stride, in bytes, is a signed 9-bit quantity that must be divisible by 2 (since it occupies only 8 bits of the instruction word). Sign-extend the stride to 32-bits and add to contents of register *pointer* to form next address. Update *pointer* with the calculated next address. ".Cn" selects circular buffer n = 0 - 2. See Note 2.

FIG. 11A

Vector Addressing Instruction Summary

Instruction	Syntax and Description
Load Byte, Pointer Increment, optional circular buffer	*LBP[.Cn] rT, (pointer)stride* Load contents of sign-extended byte addressed by register *pointer* into register rT. The stride, in bytes, is a signed 8-bit quantity. Sign-extend the stride to 32-bits and add to contents of register *pointer* to form next address. Update *pointer* with the calculated next address. ".Cn" selects circular buffer n = 0 - 2. See Note 2.
Load Byte Unsigned, Pointer Increment, optional circular buffer	*LBPU[.Cn] rT, (pointer)stride* Load contents of zero-extended byte addressed by register *pointer* into register rT. The stride, in bytes, is a signed 8-bit quantity. Sign-extend the stride to 32-bits and add to contents of register *pointer* to form next address. Update *pointer* with the calculated next address. ".Cn" selects circular buffer n = 0 - 2. See Note 2.
Store Twinword, Pointer Increment, optional circular buffer	*STP[.Cn] rT, (pointer)stride* Let *temp* = contents of register *pointer*. Store contents of register rT (which must be an even register) into word addressed by *temp*. Store contents of register rT+1 into word addressed by *temp*+4. The stride, in bytes, is a signed 11-bit quantity that must be divisible by 8 (since it occupies only 8 bits of the instruction word). Sign-extend the stride to 32-bits and add to contents of register *pointer* to form next address.Update *pointer* with the calculated next address. ".Cn" selects circular buffer n = 0 - 2. See Note 2.
Store Word, Pointer Increment, optional circular buffer	*SWP[.Cn] rT, (pointer)stride* Store contents of register rT into word addressed by register *pointer*. The stride, in bytes, is a signed 10-bit quantity that must be divisible by 4 (since it occupies only 8 bits of the instruction word). Sign-extend the stride to 32-bits and add to contents of register *pointer* to form next address. Update *pointer* with the calculated next address. ".Cn" selects circular buffer n = 0 - 2. See Note 2.
Store Halfword, Pointer Increment, optional circular buffer	*SHP[.Cn] rT, (pointer)stride* Store contents of register rT[15:00] into 16-bit halfword addressed by register *pointer*. The stride, in bytes, is a signed 9-bit quantity that must be divisible by 2 (since it occupies only 8 bits of the instruction word). Sign-extend the stride to 32-bits and add to contents of register *pointer* to form next address. Update *pointer* with the calculated next address. ".Cn" selects circular buffer n = 0 - 2. See Note 2.
Store Byte, Pointer Increment, optional circular buffer	*SBP[.Cn] rT, (pointer)stride* Store contents of register rT[07:00] into byte addressed by register *pointer*. The stride, in bytes, is a signed 8-bit quantity. Sign-extend the stride to 32-bits and add to contents of register *pointer* to form next address. Update *pointer* with the calculated next address. ".Cn" selects circular buffer n = 0 - 2. See Note 2.
Move To Radiax, User	*MTRU rT, RADREG* Move the contents of register rT to one of the User Radiax registers: cbs0 - cbs2, cbe0 - cbe2, mmd, lpc0, lpe0, lps0. This instruction has a single delay slot before the updated register takes effect.

FIG 11B-

Instruction	Syntax and Description
Move From Radiax, User	*MFRU rT, RADREG* Move the contents of the designated User Radiax register (cbs0 - cbs2, cbe0 - cbe2, mmd, lpc0, lps0, lpe0) to register rT.

Nomenclature:

rT	=	r0 - r31, and must be even for LT, ST, LTP[.Cn], STP[.Cn]
base, pointer	=	r0 - r31
stride	=	8/9/10/11-bit signed value (in bytes) for byte/halfword/word/twinword ops.
displacement	=	14-bit signed value, in bytes
RADREG	=	cbs0 - cbs2, cbe0 - cbe2, mmd, lpc0, lps0, lpe0

Notes:

1. For LTP[.Cn], LWP[.Cn], LHP(U)[.Cn], LBP(U)[.Cn], rT = *pointer* is unsupported.
2. When a circular buffer is selected, the update of the pointer register is performed according to the following algorithm, which depends on the sign of the stride and the granularity of the access. A stride exactly equal to 0 is not supported:

For LBP(U).Cn and SBP.Cn:

```
     if (stride > 0 && pointer[2:0] == 111 && pointer[31:3] == CBEn)
        then  pointer <= CBSn[31:3] || 000
else if (stride < 0 && pointer[2:0] == 000 && pointer[31:3] == CBSn)
        then  pointer <= CBEn[31:3] || 111
else          pointer <= pointer + stride.
```

For LHP(U).Cn and SHP.Cn

```
     if (stride > 0 && pointer[2:0] == 11x && pointer[31:3] == CBEn)
        then  pointer <= CBSn[31:3] || 000
else if (stride < 0 && pointer[2:0] == 00x && pointer[31:3] == CBSn)
        then  pointer <= CBEn[31:3] || 110
else          pointer <= pointer + stride.
```

For LWP.Cn and SWP.Cn

```
     if (stride > 0 && pointer[2:0] == 1xx && pointer[31:3] == CBEn)
        then  pointer <= CBSn[31:3] || 000
else if (stride < 0 && pointer[2:0] == 0xx && pointer[31:3] == CBSn)
        then  pointer <= CBEn[31:3] || 100
else          pointer <= pointer + stride.
```

For LTP.Cn and STP.Cn

```
     if (stride > 0                          && pointer[31:3] == CBEn)
        then  pointer <= CBSn[31:3] || 000
else if (stride < 0                          && pointer[31:3] == CBSn)
        then  pointer <= CBEn[31:3] || 000
else          pointer <= pointer + stride.
```

FIG. 11C

Extensions to MIPS ALU Operations

1200

Instruction	Syntax and Description
Dual Shift Left Logical Variable	*SLLV2 rD, rT, rS* The contents of rT[31:16] and the contents of rT[15:00] are independently shifted left by the number of bits specified by the low order four bits of the contents of general register rS, inserting zeros into the low order bits of rT[31:16] and rT[15:00]. For SLLV2, the high and low results are concatenated and placed in register rD. (Note that a [.S] option is not provided because this is a *logical* rather than *arithmetic* shift and thus the concept of arithmetic overflow is not relevant.)
Dual Shift Right Logical Variable	*SRLV2 rD, rT, rS* The contents of rT[31:16] and the contents of rT[15:00] are independently shifted right by the number of bits specified by the low order four bits of the contents of general register rS, inserting zeros into the high order bits of rT[31:16] and rT[15:00]. The high and low results are concatenated and placed in register rD. (Note that a [.S] option is not provided because this is a *logical* rather than *arithmetic* shift and thus the concept of arithmetic overflow is not relevant.)
Dual Shift Right Arithmetic Variable	*SRAV2 rD, rT, rS* The contents of rT[31:16] and the contents of rT[15:00] are independently shifted right by the number of bits specified by the low order four bits of the contents of general register rS, sign-extending the high order bits of rT[31:16] and rT[15:00]. The high and low results are concatenated and placed in register rD. (Note that a [.S] option is not provided because arithmetic overflow/underflow is not possible.)
Add, optional saturation	*ADDR[.S] rD, rS, rT* 32-bit addition. Considering both quantities as signed 32-bit integers, add the contents of register rS to rT. For ADDR, the result is placed in register rD, ignoring any overflow or underflow. For ADDR.S, the result is saturated to $0 \| 1^{31}$ (if overflow) or $1 \| 0^{31}$ (if underflow) then placed in rD. ADDR[.S] will not cause an Overflow Trap.
Dual Add, optional saturation	*ADDR2[.S] rD, rS, rT* Dual 16-bit addition. Considering all quantities as signed 16-bit integers, add the contents of register rS[15:00] to rT[15:00] and, independently add the contents of register rS[31:16] to rT[31:16]. For ADDR2, the high and low results are concatenated and placed in register rD ignoring any overflow or underflow. For ADDR2.S, the two results are independently saturated to $0 \| 1^{15}$ (if overflow) or $1 \| 0^{15}$ (if underflow) then placed in rD. ADDR2[.S] will not cause an Overflow Trap.

FIG. 12A

Instruction	Syntax and Description
Subtract, optional saturation	*SUBR[.S] rD, rS, rT* 32-bit subtraction. Considering both quantities as signed 32-bit integers, subtract the contents of register rT from the contents of register rS. For SUBR, the result is placed in register rD ignoring any overflow or underflow. For SUBR.S, the result is saturated to $0 \| 1^{31}$ (if overflow) or $1 \| 0^{31}$ (if underflow) then placed in rD. SUBR[.S] will not cause an Overflow Trap.
Dual Subtract, optional saturation	*SUBR2[.S] rD, rS, rT* Dual 16-bit subtraction. Considering all quantities as signed 16-bit integers, subtract the contents of register rT[15:00] from rS[15:00] and, independently subtract the contents of register rT[31:16] from rS[31:16]. For SUBR2, the high and low results are concatenated and placed in register rD ignoring any overflow or underflow. For SUBR2.S, the two results are independently saturated to $0 \| 1^{15}$ (if overflow) or $1 \| 0^{15}$ (if underflow) then placed in rD. SUBR2[.S] will not cause an Overflow Trap.
Dual Set On Less Than	*SLTR2 rD, rS, rT* Dual 16-bit comparison. Considering both quantities as signed 16-bit integers, if rS[15:00} is less than rT[15:00] then set rD[15:00] to $0^{15} \| 1$, else to zero. Independently, considering both quantities as signed 16-bit integers, if rS[31:16} is less than rT[31:16] then set rD[31:16] to $0^{15} \| 1$, else to zero.

Nomenclature:

rD	=	r0 - r31
rS	=	r0 - r31
rT	=	r0 - r31

FIG. 12B

ALU Operations

1300

Instruction	Syntax and Description
Minimum	*MIN rD, rS, rT* The contents of the general register rT are compared with rS considering both quantities as signed 32-bit integers. If rS < rT or rS = rT, rS is placed into rD. If, rS > rT, rT is placed into rD.
Dual Minimum	*MIN2 rD, rS, rT* The contents of rT[31:16] are compared with rS[31:16] considering both quantities as signed 16-bit integers. If rS[31:16] < rT[31:16] or rS[31:16] = rT[31:16], rS[31:16] is placed into rD[31:16]. If, rS[31:16] > rT[31:16], rT[31:16] is placed into rD[31:16]. A similar, independent operation is performed on rT[15:00] and rS[15:00] to determine rD[15:00].
Maximum	*MAX rD, rS, rT* The contents of the general register rT are compared with rS considering both quantities as signed 32-bit integers. If rS > rT or rS = rT, rS is placed into rD. If, rS < rT, rT is placed into rD.
Dual Maximum	*MAX2 rD, rS, rT* The contents of rT[31:16] are compared with rS[31:16] considering both quantities as signed 16-bit integers. If rS[31:16] > rT[31:16] or rS[31:16] = rT[31:16], rS[31:16] is placed into rD[31:16]. If, rS[31:16] < rT[31:16], rT[31:16] is placed into rD[31:16]. A similar, independent operation is performed on rT[15:00] and rS[15:00] to determine rD[15:00].
Absolute, optional saturation	*ABSR[.S] rD, rT* Considering rT as a signed 32-bit integer, if rT > 0, rT is placed into rD. If rT < 0, -rT is placed into rD. If ABSR.S and rT = $1 \| 0^{31}$ (the smallest negative number) then $0 \| 1^{31}$ (the largest positive number) is placed into rD; otherwise, if ABSR and rT = $1 \| 0^{31}$, rT is placed into rD.
Dual Absolute, optional saturation	*ABSR2[.S] rD, rT* ABS[.S] operations are performed independently on rT[31:16] and rT[15:00], considering each to be 16-bit signed integers. rD is updated with the absolute value of rT[31:16] concatenated with the absolute value of rT[15:00].
Dual Mux	*MUX2[[.HH], [.HL], [.LH], [.LL]] rD, rS, rT* rD[31:16] is updated with rS[31:16] for MUX2.HH or MUX2.HL. rD[31:16] is updated with rS[15:00] for MUX2.LH or MUX2.LL. rD[15:00] is updated with rT[31:16] for MUX2.HH or MUX2.LH rD[15:00] is updated with rT[15:00] for MUX2.HL or MUX2.LL
Count Leading Sign bits	*CLS rD, rT* The binary-encoded number of redundant sign bits of general register rT is placed into rD. If rT[31:30] = 10 or 01, rD is updated with 0. If rT = 0, or if rT = 1^{32}, rD is updated with $0^{27} \| 1^{5}$ (decimal 31).

FIG. 13A

ALU Operations

Instruction	Syntax and Description
Bit Reverse	*BITREV rD, rT, rS* A bit-reversal of the contents of general register rT is performed. The result is then shifted right logically by the amount specified in the lower 5-bits of the contents of general register rS, then stored in rD.

1300

Nomenclature:

rD	=	r0 - r31
rS	=	r0 - r31
rT	=	r0 - r31

FIG. 13B

Conditional Operations

1400

Instruction	Syntax and Description
Conditional Move on Equal Zero	*CMVEQZ[.H] [.L] rD, rS, rT* If the general register rT is equal to 0, the general register rD is updated with rS; otherwise rD is unchanged. For [.H] if rT[31:16] is equal to 0, the *full 32-bit* general register rD[31:00] is updated with rS; otherwise rD is unchanged. For [.L] if rT[15:00] is equal to 0, the *full 32-bit* general register rD[31:00] is updated with rS; otherwise rD is unchanged.
Conditional Move on Not Equal Zero	*CMVNEZ[.H] [.L] rD, rS, rT* If the general register rT is not equal to 0, the general register rD is updated with rS; otherwise rD is unchanged. For [.H] if rT[31:16] is not equal to 0, the *full 32-bit* general register rD[31:00] is updated with rS; otherwise rD is unchanged. For [.L] if rT[15:00] is not equal to 0, the *full 32-bit* general register rD[31:00] is updated with rS; otherwise rD is unchanged.

Nomenclature:

rD	=	r0 - r31
rS	=	r0 - r31
rT	=	r0 - r31

Usage Note:
When combined with the SLT or SLTR2 instructions, the conditional move instructions can be used to construct a complete set of conditional move macro-operations. For example:

```
if ( r3 < r4 )   r1 <-- r2

  CMVLT   r1,r2,r3,r4      ===> SLT      AT,r3,r4
                                CMVNEZ    r1,r2,AT
if ( r3 >= r4 )  r1 <-- r2

  CMVGE   r1,r2,r3,r4      ===> SLT      AT,r3,r4
                                CMVEQZ    r1,r2,AT
if ( r3 <= r4 )  r1 <-- r2

  CMVLE   r1,r2,r3,r4      ===> SLT      AT,r4,r3
                                CMVEQZ    r1,r2,AT
f( r3 > r4 )  r1 <-- r2

  CMVGT   r1,r2,r3,r4      ===> SLT      AT,r4,r3
                                CMVNEZ    r1,r2,AT
```

FIG. 14

Cycles Required Between Dual MAC Instructions

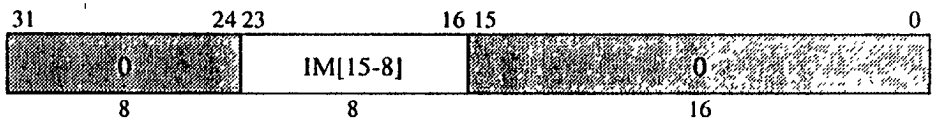
1500

2nd Op \ 1st Op	MULTA(U)	MADDA(U), MSUBA(U)	CMULTA	DIVA(U)	MADDA2[.S], MSUBA2[.S], ADDMA[.S], SUBMA[.S], MULTA2, MULNA2, RNDA2, MTA2	MFA
MULTA(U), MADDA(U),MSUBA(U)	1U	1U	1U	(19T)	-	-
DIVA(U)	(3T)	(4T)	(1T)	19U	-	-
CMULTA, MADDA2[.S], MSUBA2[.S], MULTA2, MULNA2, MTA2	3U	4U	1U	(19T)	-	-
ADDMA[.S], SUBMA[.S], RNDA2	LO: 2S, 2T HI: 3S, 3T	4U	1S, 1T	(19S), (19T)	-	-
MFA	LO: 4S HI: 5S	LO: 5S HI: 6S	3S	19S	2S	-

Notes:
- means the two ops can be issued back-to-back.
xU indicates unconditional delay of the indicated number of cycles.
xS indicates delay only if (any) 2nd Op source is the same as (any) 1st Op target (producer-consumer dependency).
xT indicates delay only if (any) 2nd Op target is the same as (any) 1s Op target (preserve write after write order).
Items in parenthesis are unlikely to occur in any useful program, which would probably have an intervening MFA.
LO/HI indicate that for the 72-bit result of a 32x32 MULT or MADDA, the LO 32-bits (m0l, m1l, etc.) are available one cycle earlier.
Delay of "x" cycles means that if the 1st Op issues in cycle N, then the 2nd Op may issue in cycle N+x+1.

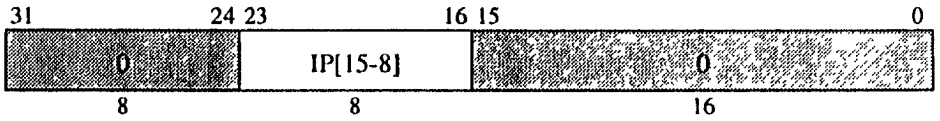
FIG.15

New: ESTATUS (LX COP0 reg 0) Read/Write



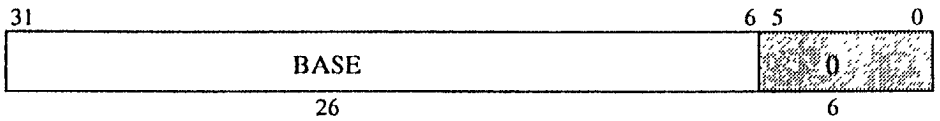
1610

New: ECAUSE (LX COP0 reg 1) Read-only



1620

New: INTVEC (LX COP0 reg 2) Read/Write

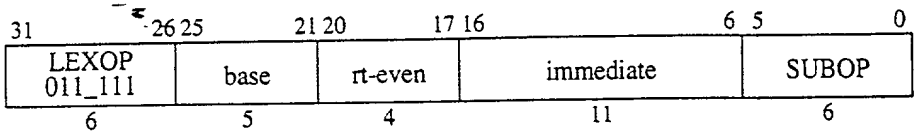


1630

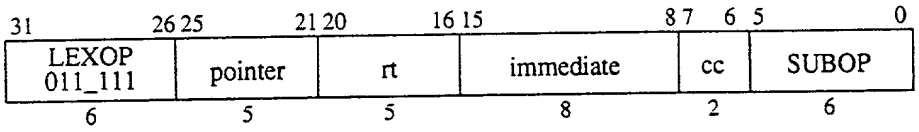
IM[15-8] is reset to 0.

FIG. 16

I. Load/Store Formats



Assembler Mnemonic	base	rt-even	immediate	Lexra SUBOP
LT	base	rt-even	displacement/8	LT
ST	base	rt-even	displacement/8	ST

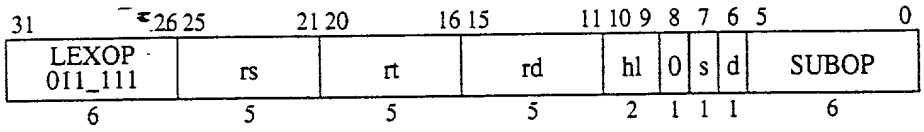


Assembler Mnemonic	pointer	rt	immediate	cc	Lexra SUBOP
LBP[.Cn]	pointer	rt	stride	cc	LBP
LBPU[.Cn]	pointer	rt	stride	cc	LBPU
LHP[.Cn]	pointer	rt	stride/2	cc	LHP
LHPU[.Cn]	pointer	rt	stride/2	cc	LHPU
LWP[.Cn]	pointer	rt	stride/4	cc	LWP
LTP[.Cn]	pointer	rt	stride/8	cc	LTP
SBP[.Cn]	pointer	rt	stride	cc	SBP
SHP[.Cn]	pointer	rt	stride/2	cc	SHP
SWP[.Cn]	pointer	rt	stride/4	cc	SWP
STP[.Cn]	pointer	rt	stride/8	cc	STP

base, pointer, rt — Selects general register r0 - r31.
rt-even — Selects general register even-odd pair r0/r1, r2/r3, ... r30/r31
stride — Signed 2s-complement number in bytes. Must be an integral number of halfwords/words/twinwords for the corresponding instructions.
displacement — Signed 2s-complement number in bytes. Must be an integral number of twinwords.
cc
00 — select circular buffer 0 (cbs0, cbe0)
01 — select circular buffer 1 (cbs1, cbe1)
10 — select circular buffer 2 (cbs2, cbe2)
11 — no circular buffer selected

FIG. 17A

II. Arithmetic Format



Assembler Mnemonic	rs	rt	rd	hl	s	d	Lexra SUBOP
ADDR[.S],ADDR2[.S]	rs	rt	rd	0	s	d	ADDR
SUBR.S, SUBR2[.S]	rs	rt	rd	0	s	d	SUBR
SLTR2	rs	rt	rd	0	0	1	SLTR
SLLV2	rs	rt	rd	0	0	1	SLLV
SRLV2	rs	rt	rd	0	0	1	SRLV
SRAV2	rs	rt	rd	0	0	1	SRAV
MIN, MIN2	rs	rt	rd	0	0	d	MIN
MAX, MAX2	rs	rt	rd	0	0	d	MAX
ABSR[.S], ABSR2[.S]	0	rt	rd	0	s	d	ABSR
MUX2.[LL,LH,HL,HH]	rs	rt	rd	hl	0	1	MUX
CLS	0	rt	rd	0	0	0	CLS
BITREV	rs	rt	rd	0	0	0	BITREV

rs, rt, rd

Selects general register r0 - r31.

s

Selects saturation of result. s=1 indicates that saturation is performed.

d

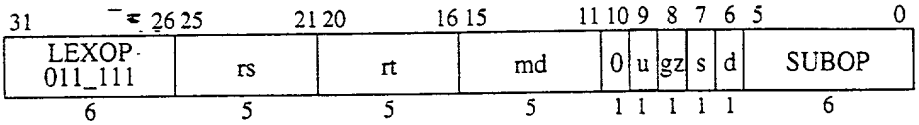
d=1 indicates that dual operations on 16-bit data are performed.

hl (for MUX2)

00 LL: rD = rs[15:00] || rt[15:00]
01 LH: rD = rs[15:00] || rt[31:16]
10 HL: rD = rs[31:16] || rt[15:00]
11 HH: rD = rs[31:16] || rt[31:16]

FIG. 17B

III. MAC Format A



Assembler Mnemonic	rs	rt	md	u	gz	s	d	Lexra SUBOP
CMULTA	rs	rt	md	0	0	0	0	CMULTA
DIVA(U)	rs	rt	md	u	0	0	0	DIVA
MULTA(U)	rs	rt	md	u	1	0	0	MADDA
MULTA2	rs	rt	md	0	1	0	1	MADDA
MADDA(U)	rs	rt	md	u	0	0	0	MADDA
MADDA2[.S]	rs	rt	md	0	0	s	1	MADDA
MSUBA(U)	rs	rt	md	u	0	0	0	MSUBA
MSUBA2[.S]	rs	rt	md	0	0	s	1	MSUBA
MULNA2	rs	rt	md	0	1	0	1	MSUBA
MTA2[.G]	rs	0	md	0	g	0	1	MTA

rs, rt — Selects general register r0 - r31.

md — Selects accumulator, 0NNHL where,

NN = m0 - m3
HL
00 = reserved — 01 = mNl
10 = mNh — 11 = mN

s — Selects saturation of result. s=1 indicates that saturation is performed.

d — d=1 indicates that dual operations on 16-bit data are performed.

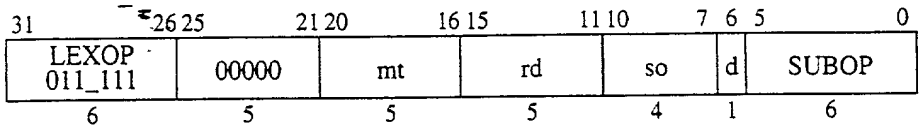
gz — For MTA2, used as "guard" bit. If g=1, bits [39:32] of the accumulator (pair) are loaded and bits [31:00] are unchanged. If g=0, all 40 bits [39:00] of the accumulator (or pair) are updated.

For MADDA, MSUBA, used as a "zero" bit. If z = 1, the result is added to (subtracted from) zero rather than the previous accumulator value; this performs a MULTA, MULTA2 or MULNA2. If z = 0, performs a MADDA, MSUBA, MADDA2 or MSUBA2.

u — Treat operands as unsigned values (0 = signed, 1 = unsigned)

FIG. 17C

IV. MAC Format B



Assembler Mnemonic	mt	rd	so	d	Lexra SUBOP
MFA, MFA2	mt	rd	so	d	MFA
RNDA2	mt	0	so	1	RNDA

rd

Selects general register r0 - r31.

mt Selects accumulator, 0NNHL where,

NN = m0 - m3
HL
00 = reserved
01 = mNl
10 = mNh
11 = mN

d

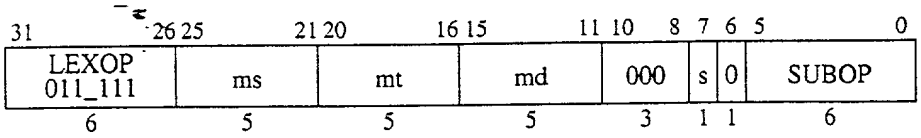
d=1 indicates that dual operations on 16-bit data are performed.

so

Encoded ("output") shift amount n = 0 - 8 for RNDA2, MFA, MFA2 instructions.

FIG. 17D

V. MAC Format C



Assembler Mnemonic	ms	mt	md	s	Lexra SUBOP
ADDMA[.S]	ms	mt	md	s	ADDMA
SUBMA[.S]	ms	mt	md	s	SUBMA

mt, ms, md

Selects accumulator, 0NNHL where,

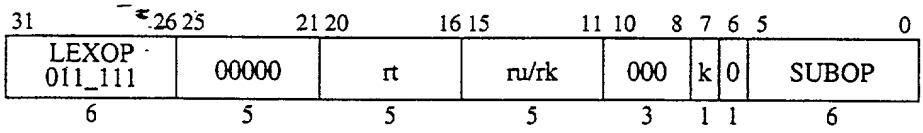
NN = m0 - m3
HL
00 = reserved
01 = mNl
10 = mNh
11 = reserved

s

Selects saturation of result. s=1 indicates that saturation is performed.

FIG. 17E

VI. RADIAX MOVE Format and Lexra-Cop0 MTLXC0/MFLXC0 Instructions



Assembler Mnemonic	rt	ru/rk	k	Lexra SUBOP
MFRU	rt	ru	0	MFRAD
MTRU	rt	ru	0	MTRAD
MFRK	rt	rk	1	MFRAD
MTRK	rt	rk	1	MTRAD

rt

Selects general register r0 - r31.

rk

Selects Radiax Kernel register in MFRK, MTRK instructions — currently all reserved. However, a Coprocessor Unusable Exception is taken in User mode if the Cu0 bit is 0 in the CP0 Status register when MFRK or MTRK is executed.

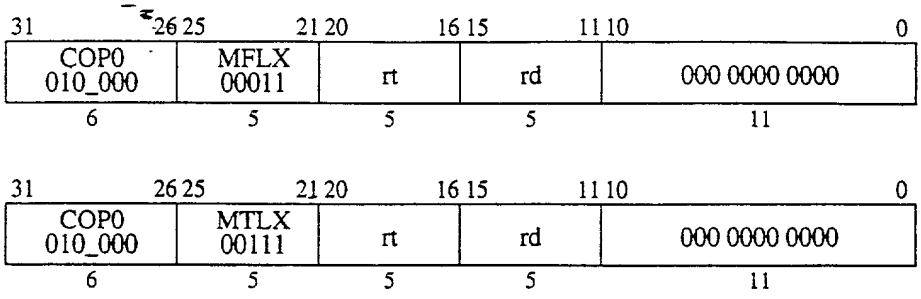
ru

Selects Radiax User register in MFRU, MTRU instructions.

00000 cbs0
00001 cbs1
00010 cbs2
00011 reserved
00100 cbe0
00101 cbe1
00110 cbe2
00111 reserved
01xxx reserved
10000 lps0
10001 lpe0
10010 lpc0
10011 reserved
101xx reserved
11000 mmd
11001 reserved
111xx reserved

FIG. 17F

Lexra-Coprocessor0 Register Access Instructions



Assembler Mnemonic	Copz rs	rt	rd
MFLXC0	MFLX	rt	rd
MTLXC0	MTLX	rt	rd

These are *not* LEXOP instructions. They are variants of the standard MTC0 and MFC0 instructions that allow access to the Lexra Coprocessor0 Registers listed below. As with any COP0 instruction, a Coprocessor Unusable Exception is taken in User mode if the Cu0 bit is 0 in the CP0 Status register when these instructions are executed.

rt

Selects general register r0 - r31.

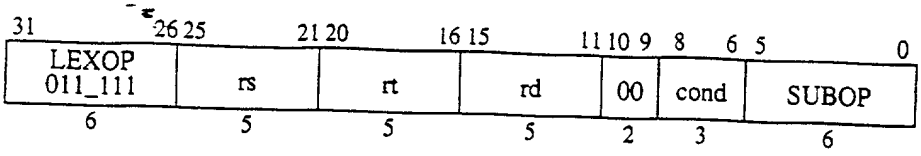
rd

Selects Lexra Coprocessor0 register:

00000 ESTATUS
00001 ECAUSE
00010 INTVEC
00011 reserved
001xx reserved
01xxx reserved
1xxxx reserved

FIG. 17G

VII. CMOVE Format



Assembler Mnemonic	rs	rt	rd	cond	Lexra SUBOP
CMVEQZ[.H][.L]	rs	rt	rd	cond	CMOVE
CMVNEZ[.H][.L]	rs	rt	rd	cond	CMOVE

rs, rt, rd

Selects general register r0 - r31.

cond

Condition code for rT operand referenced by the conditional move.

000 EQZ
001 NEZ
010 EQZ.H
011 NEZ.H
100 EQZ.L
101 NEZ.L
11x reserved

FIG. 17H